Hierarchically Semi-separable Representation and its applications

Zhifeng Sheng

Hierarchically Semi-separable Representation and its applications

Master's Thesis in Computer Engineering

Circuit and System Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Zhifeng Sheng

April 3, 2006

Author
Zhifeng Sheng
Title
Hierarchically Semi-separable Representation and its applications
MSc presentation
April 28th, 2006

Graduation Committee

Prof. dr. ir. Patrick Dewilde (chair)	Delft University of Technology
Prof. dr. ir. Alle-Jan van der Veen	Delft University of Technology
dr. ir. Nick van der Meijs	Delft University of Technology
dr. ir. Remis, RF	Delft University of Technology

Abstract

Much of the research of numerical linear algebra has been triggered by a problem that can be posed simply as: Given $A \in C^{m \times n}, b \in C^m$, find solution vector(s) $x \in C^n$. Many scientific problems lead to the requirement to solve a linear system of equations or multiply a system matrix with a vector as part of computations. With the ever increasing size of the system matrix, the straightforward matrix operation algorithms run quickly out of steam. To accelerate matrix operations, not only the entry sparsity but also the data sparsity of the system matrix should be utilized.

In this thesis, we study a important class of structured matrices: "Hierarchically Semi-Separable" matrices, for which an efficient hierarchically state based representation called Hierarchically Semi-Separable (HSS) representation can be used to represent it with a small amount of parameters that are linear in the dimension of the matrix. Under the framework of this HSS representation, efficient matrix transformation algorithms that are linear in the number of equations are given. In particular, a system $Ax = b$ can be solved with linear complexity. Also, LU and URV factorization can be efficiently executed. There is a close connection between HSS matrices and classical Semi Separable matrices (sometimes called 'Sequentially Semi Separable matrices or SSS matrices or equivalently, time-varying systems), and a number of results of SSS theory can be translated to parallel results in the HSS theory (neither class contains the other however). We also discuss the limitation of the direct HSS algorithms, for which iterative solution algorithms based on HSS representation and its basic algorithms should come to rescue. A number of *preconditioner* construction algorithms based on HSS representation have been proposed.

As a somewhat separated topic, we implement the Integrated Field Equations Method on triangular finite element mesh to simulate the electromagnetic field in case of perpendicular polarization. A simple 2D case is tested and the system matrix has been generated.

To evaluate all these algorithms, we experiment the behavior of these algorithms extensively on series of randomly generated HSS matrices and on practical system matrices from the *MatrixMarket*. Our experiments suggest that all these algorithms have linear complexity in computation time and more importantly in memory usage.

Acknowledgements

I would like to thank prof. Patrick dewilde, my supervisor in TU Delft Circuit and System group. He gave me unconditional support in many aspects during the whole process. Most importantly he encouraged me through the whole process. I also would like to thank prof. Nick Van der Meijs in TU Delft Circuit and System group, whom I always discussed with when I had any problems regarding the background information. Without their help, the whole project and this thesis would not become true.

Thanks to all my friends who supported me with this thesis: Han Wang, whom I always discussed with and shared my spare time with; Keesjan Van der Kolk, whom I learned a lot from.

I would also like to thank Georgi Gaydadjiev, my programmer coordinator, and all the professors in Computer engineering group who have given me many supports during the two-year program here in Delft.

Thanks to all my friends in the Netherlands and China, whom I can share my joy and sorrow with.

Finally, thanks to my beloved parents for their endless love and support.

Zhifeng Sheng

Delft, The Netherlands
April 3, 2006

Contents

List of Figures

Chapter 1

Introduction

1.1 Semi-separble systems

The term 'semi-separable systems' originated in the work of Gohberg, Kailath and Koltracht [14] where these authors remarked that if an integral kernel is approximated by an outer sum, then the system could be solved with a number of operations essentially determined by the order of the approximation rather than by a power of the number of input and output data. In the same period, Greengard and Rokhlin [15, 29] proposed the 'multipole method' where an integral kernel such as a Green's function is approximated by an outer product resulting in a matrix in which large sub-matrices have low rank. These two theories evolved in parallel in the system theoretical literature and the numerical literature. In the system theoretical literature it was realized that an extension of the semi-separable model (sometimes called 'quasi-separability') brings the theory into the realm of time-varying systems, with its rich theory of state realization, interpolation, model order reduction, factorization and embedding [9]. In particular, it was shown in [34] that, based on this theory, a numerically backward stable solver of low complexity can be derived realizing a URV factorization of an operator T, in which U and V are low unitary matrices of state dimensions at most as large as those of T and R is causal, outer and also of state dimensions at most equal those of T. Subsequently, this approach has been refined by a number of authors, a.o. [11, 5, 10].

Although the SSS theory leads to very satisfactory results, it also became apparent in the late nineties that it is insufficient to cover major physical situations in which it would be very helpful to have system solvers of low complexity - in which of the often very large size of the matrices involved. Is it possible to extend the framework of SSS systems so that its major properties remain valid, in particular the fact that the class is closed under system inversion. The HSS theory, pioneered by Chandrasekaran and Gu [6] provides an answer to this question. It is based on a different state space model than the SSS theory, namely a hierarchical rather than a sequential one, but it handles the transition operators very much in the same taste. Based on this a theory that is very much in the same flavor as the basic time-varying theory of [9] can be developed, and many results carry over. In the remainder of this thesis we show how this comes about, derive some major results concerning system inversion, and discuss some further perspectives. The remainder sections of this introduction are devoted to a brief summary of the construction of SSS systems which lay at the basis of the HSS theory. In the numerical literature, the efforts have been concentrated on 'smooth' matrices, i.e. matrices in which large sub matrices can be approximated by low rank matrices thanks to the fact that their entries are derived from smooth kernels [18, 26]. Both the SSS and HSS structures are more constrained than the 'H-matrices' considered by Hackbusch a.o., but they do have the desirable property that they are closed under inversion and fit

naturally in a state space framework. In this thesis we explore in particular the state space structure of HSS systems, other structures such as hierarchical multi-band decomposition have also been considered [8] but are beyond the present scope.

Our basic context is that of block matrices or operators $T = [T_{i,j}]$ with rows of dimensions $\cdots, m_{-1}, m_0, m_1, \cdots$ and column dimensions $\cdots, n_{-1}, n_0, n_1, \cdots$. Any of these dimensions may be zero, resulting in an empty row or column (matrix calculus can easily be extended to cover this case, the main rule being that the product of a matrix of dimensions $m \times 0$ with a matrix of dimensions $0 \times n$ results in a zero matrix of dimensions $m \times n$). Concentrating on an upper block matrix (i.e. when $T_{i,j} = 0$ for $i > j$), we define the *the degree of semi-separability* of T as the sequence of ranks $[\delta_i]$ of the matrices H_i where H_i is the sub-matrix corresponding to the row indexes $\cdots, n_{i-2}, n_{i-1}$ and the column indexes $m_i, m_{i+1}, \cdots$. H_i is called the i^{th} Hankel operator of the matrix T. In case of infinite dimensional operators (as is often the case in system theory), we say that the system is *locally finite* if all H_i have finite dimensions. Corresponding to the local dimension δ_i there are minimal factorizations $H_i = \mathcal{C}_i\mathcal{O}_i$ into what are called the i^{th} *controllability matrix* $\mathcal{C}_i$ and *observability matrix* $\mathcal{O}_i$, of dimensions $\cdot \times \delta_i$ and $\delta_i \times \cdot$. Connected to such a system of factorizations there is an indexed realization $\{A_i, B_i, C_i, D_i\}$ of dimensions $\{\delta_i \times \delta_{i+1}, m_i \times \delta_{i+1}, \delta_i \times n_i, m_i \times n_i\}$ constituting a local set of 'small' matrices with the characteristic property of semi-separable realizations for which it holds that

$$\left\{ \begin{array}{ccc} \mathcal{C}_i = \left[\begin{array}{c} \vdots \\ B_{i-2}A_{i-1} \\ B_{i-1} \end{array} \right] & , & \mathcal{O}_i = \left[\begin{array}{cccc} C_i & A_iC_{i+1} & A_iA_{i+1}C_{i+2} & \cdots \end{array} \right] \\ T_{i,j} = D_i & \text{for} & i = j \\ T_{i,j} = B_iA_{i+1}\cdots A_{j-1}C_j & \text{for} & i < j. \end{array} \right. \tag{1.1}$$

The vector-matrix multiplication $y = uT$ can be represented by local state space computations

$$\left\{ \begin{array}{rcl} x_{i+1} & = & x_iA_i + u_iB_i \\ y_i & = & x_iC_i + u_iD_i \end{array} \right. \tag{1.2}$$

The goal of most semi-separable computational theory (as done in [9]) is to perform computations with a complexity linear in the overall dimensions of the matrix, and some function of the degree δ_i, preferably linear, but that is often not achievable (there is still quite some work to do on this topic even in the SSS theory!). The above briefly mentioned realization theory leads to nice representations of the original operator. To this end we only need to introduce a shift operator Z with the characteristic property $Z_{i,i+1} = I$, zero elsewhere, where the dimension of the unit matrix is context dependent, and global representations for the realization as block diagonal operators $\{A = \text{diag}[A_i], B = \text{diag}[B_i], C = \text{diag}[C_i], D = \text{diag}[D_i]\}$. The lower triangular part can of course be dealt with in the same manner as the upper, resulting in the general semi-separable representation of an operator as the superscript 'H' indicates Hermitian conjugation)

$$T = B_\ell Z^H (I - A_\ell Z^H)^{-1} C_\ell + D + B_u Z (I - A_u Z)^{-1} C_u \tag{1.3}$$

in which the indexes refer to the lower, respect. upper semi-separable decomposition. In general we assume that the inverses in this formula do exist and have reasonable bounds, if that is not the case one has to resort to different techniques that go beyond the present exposition. In the finite dimensional case the matrix $(I - AZ)$ takes the special form when the indexing runs from 0 to n (for orientation the $0, 0$

element is boxed in):

$$(I - AZ) = \begin{bmatrix} \boxed{I} & A_0 & & & \\ & I & A_1 & & \\ & & \ddots & \ddots & \\ & & & I & A_n \\ & & & & I \end{bmatrix} \tag{1.4}$$

one may think that this matrix is always invertible, but that is numerically not true, how to deal with numerical instability in this context is also still open territory.

The SSS theory (alias time-varying system theory) has produced many results paralleling the classical LTI theory and translating these results to a matrix context, (see [9] for a detailed account):

- *System inversion*: $T = URV$ in which the unitary matrices U, V and the outer matrix R (outer means: upper and upper invertible) are all semi-separable of degree at most the degree of T;
- *System approximation and model reduction*: sweeping generalizations of classical interpolation theory of the types Nevanlinna-Pick, Caratheodory-Fejer and even Schur-Takagi, resulting in a complete model reduction theory of the 'AAK-type' but now for operators and matrices;
- *Cholesky and spectral factorization*: $T = FF^*$ when T is a positive operator, in which F is semi-separable of the same degree sequence as T - a theory closely related to Kalman filtering;
- and many more results in embedding theory and minimal algebraic realization theory.

1.2 Hierarchical semi-separable systems

The Hierarchical Semi-Separable representation of a matrix (or operator) A is a layered representation of the multi-resolution type, indexed by the hierarchical level. At the top level 1, it is a 2×2 block matrix representation of the form (notice the redefinition of the symbol A):

$$A = \begin{bmatrix} A_{1;1,1} & A_{1;1,2} \\ A_{2;2,1} & A_{2;2,2} \end{bmatrix} \tag{1.5}$$

in which we impolitely assume that the ranks of the off-diagonal blocks is low so that they can be represented by an 'economical' factorization ('H' indicates Hermitian transposition, for real matrices just transposition), as follows:

$$A = \begin{bmatrix} D_{1;1} & U_{1;1}B_{1;1,2}V_{1;2}^H \\ U_{1;2}B_{1;2,1}V_{1;1}^H & D_{1;2} \end{bmatrix} \tag{1.6}$$

The second hierarchical level is based on a further but similar decomposition of the diagonal blocks, respect. $D_{1;1}$ and $D_{1;2}$:

$$D_{1;1} = \begin{bmatrix} D_{2:1} & U_{2;1}B_{2;1,2}V_{2;2}^H \\ U_{2;2}B_{2;2,1}V_{2;1}^H & D_{2;2} \end{bmatrix}, \; D_{1;2} = \begin{bmatrix} D_{2;3} & U_{2;3}B_{2;3,4}V_{2;4}^H \\ U_{2;4}B_{2;4,3}V_{2;3}^H & D_{2;4} \end{bmatrix} \tag{1.7}$$

for which we have the further *level compatibility* assumption

$$\mathrm{span}(U_{1;1}) \subset \mathrm{span}\left(\begin{bmatrix} U_{2;1} \\ 0 \end{bmatrix} \right) \oplus \mathrm{span}\left(\begin{bmatrix} 0 \\ U_{2;2} \end{bmatrix} \right), \tag{1.8}$$

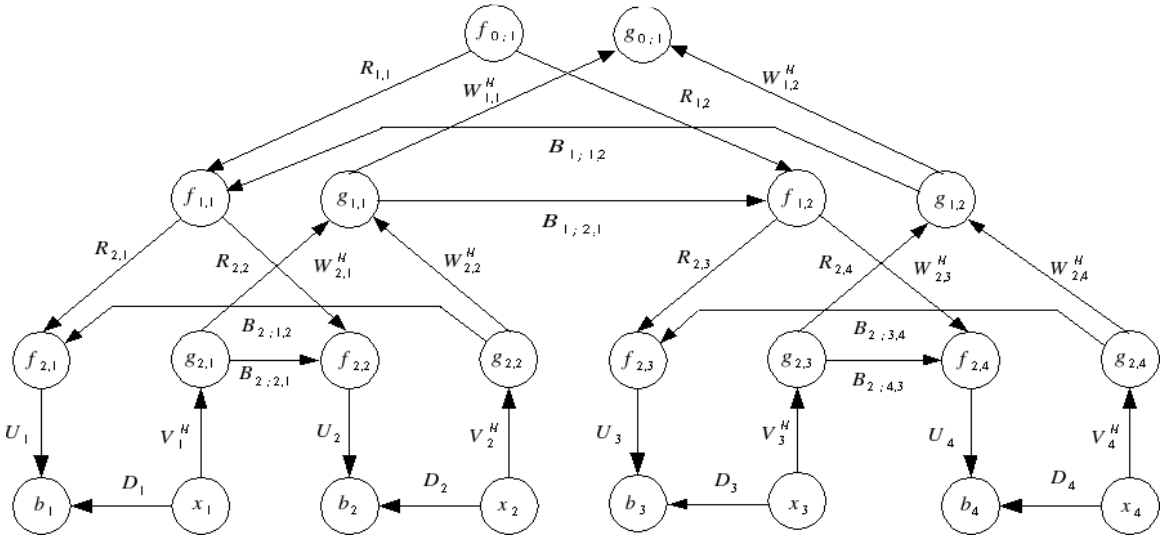

Figure 1.1: HSS Data-flow diagram for a two level hierarchy representing operator-vector multiplication, arrows indicate matrix-vector multiplication of sub-data, nodes correspond to states and are summing incoming data (the top levels f_0 and g_0 are empty).

$$\text{span}(V_{1;1}) \subset \text{span}\left(\begin{bmatrix} V_{2;1} \\ 0 \end{bmatrix}\right) \oplus \text{span}\left(\begin{bmatrix} 0 \\ V_{2;2} \end{bmatrix}\right) \text{ etc...} \tag{1.9}$$

This spanning property is characteristic for the HSS structure, it allows for a substantial improvement on the numerical complexity for e.g. vector-matrix multiplication as a multiplication with the higher level structures always can be done using lower level operations, using the *translation operators*

$$U_{1;i} = \begin{bmatrix} U_{2;2i-1}R_{2;2i-1} \\ U_{2;2i}R_{2;2i} \end{bmatrix}, \quad i = 1, 2, \tag{1.10}$$

$$V_{1;i} = \begin{bmatrix} V_{2;2i-1}W_{2;2i-1} \\ V_{2;2i}W_{2;2i} \end{bmatrix}, \quad i = 1, 2. \tag{1.11}$$

Notice the use of indexes: at a given level i rows respect. columns are subdivided in blocks indexed by $1, \cdots, i$. Hence the ordered index $(i; k, \ell)$ indicates a block at level i in the position (k, ℓ) in the original matrix. The same kind of subdivision can be used for column vectors, row vectors and bases thereof (as are generally represented in the matrices U and V).

In [4] it is shown how this multilevel structure leads to efficient matrix-vector multiplication and a set of equations that can be solved efficiently as well. For the sake of completeness we review this result briefly. Let us assume that we want to solve the system $Tx = b$ and that T has an HSS representation with deepest hierarchical level K. We begin by accounting for the matrix-vector multiplication Tx. At the leave node $(K; i)$ we can compute

$$g_{K;i} = V^H_{K;i}x_{K;i}.$$

If $(k; i)$ is not a leaf node, we can infer, using the hierarchical relations

$$g_{k;i} = V^H_{k;i}x_{k;i} = W^H_{k+1;2i-1}g_{k+1;2i-1} + W^H_{k+1;2i}g_{k+1;2i}.$$

These operations update a 'hierarchical state' $g_{k;i}$ upward in the tree. To compute the result of the multiplication, a new collection of state variables $\{f_{k;i}\}$ is introduced for which it holds that

$$b_{k;i} = T_{k;i,i} + U_{k;i} f_{k;i}$$

and which can also be computed recursively downward by the equations

$$\begin{bmatrix} f_{k+1;2i-1} \\ f_{k+1;2i} \end{bmatrix} = \begin{bmatrix} B_{k+1;2i-1,2i} g_{k+1;2i} + R_{k+1;2i-1} f_{k,i} \\ B_{k+1;2i,2i-1} g_{k+1;2i-1} + R_{k+1;2i} f_{k;i} \end{bmatrix},$$

the starting point being $f_{0;} = []$, an empty matrix. At the leaf level we can now compute (at least in principle - as we do not know x) the outputs from

$$b_{K;i} = D_{K;i} x_{K;i} + U_{K;i} f_{K;i}.$$

The next step is to represent the multiplication recursions in a compact form using matrix notation and without indexes. We fix the maximum order K as before. Next we define diagonal matrices containing the numerical information, in breadth first order:

$$\mathbf{D} = \mathrm{diag}[D_{K;i}]_{i=1,\cdots,K},\ \mathbf{W} = \mathrm{diag}[(W_{1;i})_{i=1,2}, (W_{2;i})_{i=1\cdots 4}, \cdots],\ \text{etc...} \tag{1.12}$$

Next, we need two shift operators relevant for the present situation, much as the shift operator Z in time-varying system theory explained above. The first one is the shift-down operator $Z_\downarrow$ on a tree. It maps a node in the tree on its children and is a nilpotent operator. The other one is the level exchange operator $Z_\leftrightarrow$. At each level it is a permutation that exchanges children of the same node. Finally, we need the leaf projection operator $\mathbf{P}_{\text{leaf}}$ which on a state vector which assembles in breadth first order all the values $f_{k;i}$ produces the values of the leaf nodes (again in breadth first order). The state equations representing the efficient multiplication can now be written as

$$\left\{ \begin{array}{rcccc} \mathbf{g} & = & \mathbf{P}_{\text{leaf}}^H \mathbf{V}^H \mathbf{x} & + & Z_\downarrow^H \mathbf{W}^H \mathbf{g} \\ \mathbf{f} & = & \mathbf{R} Z_\downarrow \mathbf{f} & + & \mathbf{B} Z_\leftrightarrow \mathbf{g} \end{array} \right. \tag{1.13}$$

while the 'output' equation is given by

$$\mathbf{b} \ = \ \mathbf{D}\mathbf{x} \ + \ \mathbf{U}\mathbf{P}_{\text{leaf}}\mathbf{f} \ . \tag{1.14}$$

This is the resulting HSS state space representation that parallels the classical SSS state space formulation reviewed above. Written in terms of the hidden state space quantities we find

$$\begin{bmatrix} (I - Z_\downarrow^H \mathbf{W}^H) & 0 \\ -B Z_\leftrightarrow \mathbf{g} & (I - R Z_\downarrow) \end{bmatrix} \begin{bmatrix} \mathbf{g} \\ \mathbf{f} \end{bmatrix} = \begin{bmatrix} \mathbf{P}_{\text{leaf}}^H \mathbf{V}^H \\ 0 \end{bmatrix} \mathbf{x} \tag{1.15}$$

The state quantities can always be eliminated in the present context as $(I - \mathbf{W} Z_\downarrow)$ and $(I - \mathbf{R} Z_\downarrow)$ are invertible operators due to the fact that $Z_\downarrow$ is nilpotent. We obtain as a representation for the original operator

$$T\mathbf{x} = (\mathbf{D} + \mathbf{U}\mathbf{P}_{\text{leaf}}(I - \mathbf{R} Z_\downarrow)^{-1} \mathbf{B} Z_\leftrightarrow (I - Z_\downarrow^H \mathbf{W}^H)^{-1} \mathbf{P}_{\text{leaf}}^H \mathbf{V}^H)\mathbf{x} = \mathbf{b}. \tag{1.16}$$

Chapter 2

Hierarchically Semi-Separable Representation Construction

This chapter presents various algorithms to compute the HSS representations. Section 2.1 presents the Bottom-up HSS construction algorithm. Section 2.2 presents a Top-down HSS construction algorithm.

2.1 Bottom-up Hss construction algorithm

In this section we consider the problem of constructing the hierarchically semi-separable representation of a matrix given the partitioning sequence $\{m_i\}_{i=1}^n$ in a bottom up fashion. For simplicity we assume that $n = 2^k$, the algorithm can be generalized to the case that n is any integer. The construction algorithm presented can be applied to any unstructured matrix, thus it provides a way to prove the fact that any matrix has hierarchically semi-separable representation of any level. Let A be the matrix for which we want to construct the hieratically semi-separable structure according to the partition sequence $\{m_i\}_{i=1}^n$, where $\sum_{i=1}^n m_i = N$, N is the dimension of the matrix; with the partition sequence $\{m_i\}_{i=1}^n$, we partition the matrix A into $n \times n$ square blocks (note that, in this thesis, we assume matrix A to be square, while this algorithm can be easily generalized to the rectangular matrices, which are related to overdetermined or underdetermined systems.). The partition of A is presented as follows (k represents the level of partitioning):

$$A = \begin{bmatrix} A_{k;1,1} & A_{k;1,2} & ..... & A_{k;1,n-1} & A_{k;1,n} \\ A_{k;2,1} & A_{k;2,2} & ..... & A_{k;2,n-1} & A_{k;2,n} \\ ... & ... & ..... & ... & ... \\ A_{k;i,1} & A_{k;i,2} & ..... & A_{k;i,n-1} & A_{k;i,n} \\ ... & ... & ..... & ... & ... \\ A_{k;n,1} & A_{k;n,2} & ..... & A_{k;n,n-1} & A_{k;n,n} \end{bmatrix} \tag{2.1}$$

Note that, the equation above partitions the matrix for k level HSS representations. To present the idea of hierarchical partition, we define that:

$$A_{k-1;i;j} = \begin{bmatrix} A_{k;2i-1,2j-1} & A_{k;2i-1,2j} \\ A_{k;2i,2j-1} & A_{k;2i,2j} \end{bmatrix} \tag{2.2}$$

We first demonstrate the algorithm on the two level HSS, after which we will give a formal version of this algorithm.

2.1.1 Two level HSS construction

In this section we present the algorithm by constructing a two-level HSS representation. Note that, we only consider the correctness rather than the efficiency of the algorithm in this subsection. Actually a straightforward implementation of this algorithm could be too inefficient to be use in practice. We shall present some comments on how avoid redundant computation in the later sections.

The two level HSS representation will be based on the following partition:

$$A = \begin{bmatrix} \begin{bmatrix} A_{2;1,1} & A_{2;1,2} \\ A_{2;2,1} & A_{2;2,2} \end{bmatrix} & A_{1;1,2} \\ A_{1;2,1} & \begin{bmatrix} A_{2;3,3} & A_{2;3,4} \\ A_{2;4,3} & A_{2;4,4} \end{bmatrix} \end{bmatrix} \tag{2.3}$$

where

$$A_{1;1,2} = \begin{bmatrix} A_{2;1,3} & A_{2;1,4} \\ A_{2;2,3} & A_{2;2,4} \end{bmatrix} \tag{2.4}$$

According to the notation in [5], the two level HSS representation of the matrix above will be the following:

$$A = \begin{bmatrix} D_{2,1} & U_{2,1}B_{2;1,2}V_{2,2}^H & U_{2,1}R_{2,1}B_{1;1,2}W_{2,3}^H V_{2,3}^H & U_{2,1}R_{2,1}B_{1;1,2}W_{2,4}^H V_{2,4}^H \\ U_{2,2}B_{2;2,1}V_{2,1}^H & D_{2,2} & U_{2,2}R_{2,2}B_{1;1,2}W_{2,3}^H V_{2,3}^H & U_{2,2}R_{2,2}B_{1;1,2}W_{2,4}^H V_{2,4}^H \\ U_{2,3}R_{2,3}B_{1;2,1}W_{2,1}^H V_{2,1}^H & U_{2,3}R_{2,3}B_{1;2,1}W_{2,2}^H V_{2,2}^H & D_{2,3} & U_{2,3}B_{2;3,4}V_{2,4}^H \\ U_{2,4}R_{2,4}B_{1;2,1}W_{2,1}^H V_{2,1}^H & U_{2,4}R_{2,4}B_{1;2,1}W_{2,2}^H V_{2,2}^H & U_{2,4}B_{2;4,3}V_{2,2}^H & D_{2,4} \end{bmatrix}$$

It is obvious that $D_{2,i} = A_{i,i}$,Then by looking into this equation we can see immediately that $U_{2,i}$ spans the column space of the sub-matrix:

$$M_{2,i} = \begin{bmatrix} A_{2;i,1} & ... & A_{2;i,i-1} & A_{2;i,i+1} & ... & A_{2;i,4} \end{bmatrix} i = 1, 2, 3, 4 \tag{2.5}$$

A SVD decomposition can be used to get its column basis, so we let

$$M_{2,i} \approx E_{2,i} S_{2,i} F_{2,i}^H \tag{2.6}$$

denote a low rank SVD decomposition of $M_{2,i}$, Then we have $U_{2,i} = E_{2,i}$. For the same reason, we can see that $V_{2,i}$ spans the row space of the sub-matrix:

$$N_{2,i} = \begin{bmatrix} A_{2;1,i} \\ ... \\ A_{2;i-1,i} \\ A_{2;i+1,i} \\ ... \\ A_{2;4,i} \end{bmatrix} \tag{2.7}$$

A SVD decomposition can be used to get its row basis, so we let

$$N_{2,i} \approx \hat{E}_{2,i} \hat{S}_{2,i} \hat{F}_{2,i}^H \tag{2.8}$$

, then we let $V_{2,i} = \hat{F}_{2,i}$. Now we know $U_{2,i}$ and $V_{2,i}$. The next step is to compute $B_{2;i,j}$. Let $B_{2;i,j} = U_{i,j}^H A_{2;i,j} V_{i,j}$.By definition in [5], we have to pick up the translation operators $R_{2;i}, W_{2;i}$ such that:

$$U_{1,i} = \begin{bmatrix} U_{2;2i-1}R_{2;2i-1} \\ U_{2;2i}R_{2;2i} \end{bmatrix}, V_{1,i} = \begin{bmatrix} V_{2;2i-1}W_{2;2i-1} \\ V_{2;2i}W_{2;2i} \end{bmatrix}. \tag{2.9}$$

These $U_{1;i}$ and $V_{1;i}$ are not part of the two level HSS tree representation; according to the recursive definition, $U_{1;i}$should span the column space of

$$M_{1,i} = \begin{bmatrix} A_{1;i,1} & ... & A_{1;i,i-1} & A_{1;i,i+1} & ... & A_{1;i,2} \end{bmatrix} \tag{2.10}$$

for instance,$M_{1,1} = A_{1;1,2}$, So We could let $M_{1,i} = E_{1,i}S_{1,i}F_{1,i}^H$ be the low rank SVD decomposition of $M_{1,i}$. Then

$$U_{1,i} = \begin{bmatrix} U_{2;2i-1}R_{2;2i-1} \\ U_{2;2i}R_{2;2i} \end{bmatrix} = E_{1,i} = \begin{bmatrix} E_{1,i,1} \\ E_{1,i,2} \end{bmatrix} \tag{2.11}$$

let $E_{1,i,1}$ and $E_{1,i,2}$ be the appropriate partition of $E_{1,i}$, we have $U_{2;2i-1}R_{2;2i-1} = E_{1,i,1}$, $U_{2;2i}R_{2;2i} = E_{1,i,2}$, We can easily solve them and obtain $R_{2;2i-1} = U_{2;2i-1}^H E_{1,i,1}$, $R_{2;2i} = U_{2;2i}^H E_{1,i,2}$.Similarly, $V_{1;i}$ should span the row space of

$$N_{1,i} = \begin{bmatrix} A_{1;1,i} \\ ... \\ A_{1;i-1,i} \\ A_{1;i+1,i} \\ ... \\ A_{1;2,i} \end{bmatrix} \tag{2.12}$$

Again we take $N_{1,1} = A_{1;2,1}$ for example, let $N_{1,i} = \hat{E}_{1,i}\hat{S}_{1,i}\hat{F}_{1,i}^H$ be the low rank SVD decomposition of $N_{1,i}$. Then

$$V_{1,i} = \begin{bmatrix} V_{2;2i-1}W_{2;2i-1} \\ V_{2;2i}W_{2;2i} \end{bmatrix} = \hat{F}_{1,i} = \begin{bmatrix} \hat{F}_{1;i,1} \\ \hat{F}_{1;i,2} \end{bmatrix} \tag{2.13}$$

let $\hat{F}_{1;i,1}$ and $\hat{F}_{1;i,2}$ be the appropriate partition of $\hat{F}_{1;i}$, we have $V_{2;2i-1}W_{2;2i-1} = \hat{F}_{1;i,1}$, $V_{2;2i}W_{2;2i} = \hat{F}_{1;i,2}$, Again, We can solve them and obtain $W_{2;2i-1} = V_{2;2i-1}^H \hat{F}_{1;i,1}$, $W_{2;2i} = V_{2;2i}^H \hat{F}_{1;i,2}$. After computing all these Us and Vs, we can compute $B_{1;i,j}$ as $B_{1;i,j} = E_{1;i}^H A_{1;i,j} \hat{F}_{1;j}$. Then we get the two level HSS for our matrix.

2.1.2 The Formal Algorithm for constructing HSS

As we promised, a formal algorithm shall be presented for any HSS representation of any possible level. The algorithm will be given in a recursive way. Given the partition sequence $\{m_i\}_{i=1}^n$. For simplicity we assume that $n = 2^k$. The following formulas are given to compute the k level HSS representation of matrix A.

1. Let:$D_{k,i} = A_{k;i,i}$
2. Let:$M_{k,i} = \begin{bmatrix} A_{k;i,1} & ... & A_{k;i,i-1} & A_{k;i,i+1} & ... & A_{k;i,n} \end{bmatrix}$ $M_{k,i} = E_{k,i}S_{k,i}F_{k,i}^H$ be the SVD decomposition of $M_{k,i}$. Then $U_{k;i} = E_{k,i}$.

3. Let: $N_{k,i} = \begin{bmatrix} A_{k;1,i} \\ \cdots \\ A_{k;i-1,i} \\ A_{k;i+1,i} \\ \cdots \\ A_{k;n,i} \end{bmatrix}$

   $N_{k,i} = \hat{E}_{k,i}\hat{S}_{k,i}\hat{F}^H_{k,i}$ be the SVD decomposition of $N_{k,i}$.Then $V_{k;i} = \hat{F}_{k,i}$.

4. Let: $B_{k;i,j} = E^H_{k,i}A_{i,j}\hat{F}_{k,j}$.

5. Let: $U_{k-1;i} = \begin{bmatrix} U_{k-1;i,1} \\ U_{k-1;i,2} \end{bmatrix}$ then $R_{k;2i-1} = U^H_{k;2i-1}U_{k-1;i,1}, R_{k;2i} = U^H_{k;2i}U_{k-1;i,2}$.

6. let: $V_{k-1;i} = \begin{bmatrix} V_{k-1;i,1} \\ V_{k-1;i,2} \end{bmatrix}$ then $W_{k;2i-1} = V^H_{k;2i-1}V_{k-1;i,1}, W_{k;2i} = V^H_{k;2i}V_{k-1;i,2}$.

2.1.3 Remarks on the recursive formulas

1. By applying these recursive formulas, we can get the HSS representation of any level for any matrices . However, without a method to compute SVD decomposition of a larger block from the SVD decompositions of its sub-matrices, the computation complexity of this algorithm will be $O(N^3)$. While by avoiding redundant computation on these SVD decomposition, we may get a construction algorithm of computation complexity $O(N^2)$.

2. Even better, for some practical cases, we could make use of the structure of the system[1]. And for many cases, we expect efficient construction algorithms of a computation complexity less than $O(N^2)$.

3. The reason to for us to use SVD in this paper is that it generates U_i,V_i with orthonormal columns which guarantee a HSS in a proper form(to be described in chapter 3). It is also rank revealing decomposition which could be useful for model reduction. One obvious disadvantage of this factorization is its complexity. We can substitute SVD decomposition with other factorizations such as QR factorization, ULV factorization. Whatever factorizations we use, it should be able to reveal economic row basis or column basis or both of a matrix.

2.2 Top-down Hss construction algorithm

The HSS representation can also be built from the root. Once the root is constructed, its leaves can be split recursively until certain criteria is satisfied.

In order to develop the top-down HSS construction algorithm, we need to preview the leaf-split algorithm which will be presented in the coming chapter.

[1]For example, if the system matrix comes from 1D fast multiple method, its HSS representation comes without much cost

2.2.1 Leaf-split

One big leaf could be split into two smaller leaves, while the intermediate variables f and g (to be defined) of the big leaf will become the intermediate variables of the parent of the two new leaves. Leaf split modification can be done on any leaf. Its Dataflow modification is given as below (Sorry for introducing the dataflow diagram without explanation. we will see these diagrams again in the coming chapter).

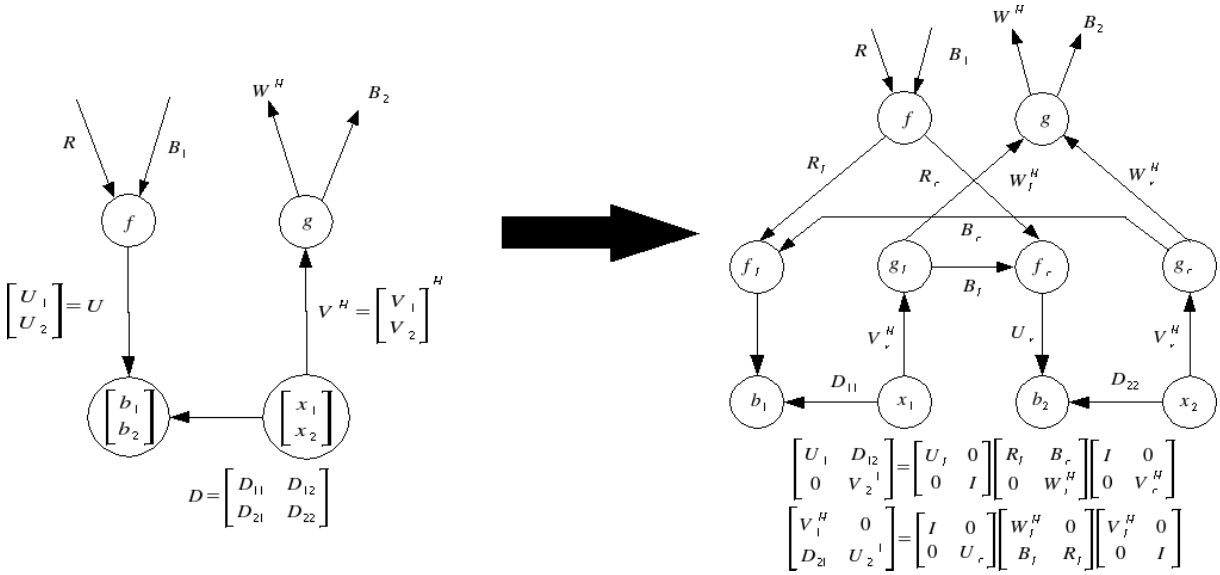

Figure 2.1: Leaf split

The formula to generate the matrices of the new leaves from those of the big one is given below:

1. Firstly, we partition the matrices in the following way. Note that the dimensions of the partitions should be appropriate, so that the following calculations below are legal. Let:

$$U = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix}, V^H = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}^H, b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, D = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} \tag{2.14}$$

2. Then, the matrices in the right hand dataflow diagram should hold the following equations (see lemma 1):

$$\begin{bmatrix} U_1 & D_{12} \\ 0 & V_2^H \end{bmatrix} = \begin{bmatrix} U_l & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} R_l & B_r \\ 0 & W_r^H \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & V_r^H \end{bmatrix} = \begin{bmatrix} U_l R_l & U_l B_r V_r^H \\ 0 & W_r^H V_r^H \end{bmatrix} \tag{2.15}$$

$$\begin{bmatrix} V_1^H & 0 \\ D_{21} & U_2 \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & U_r \end{bmatrix} \begin{bmatrix} W_l^H & 0 \\ B_l & R_r \end{bmatrix} \begin{bmatrix} V_l^H & 0 \\ 0 & I \end{bmatrix} = \begin{bmatrix} W_l^H V_l^H & 0 \\ U_r B_l V_l^H & U_r R_r \end{bmatrix} \tag{2.16}$$

Note that: U_i, V_i should be chosen such that they are column orthonormal. $\begin{bmatrix} R_{l;i} \\ R_{r;i} \end{bmatrix}$ and $\begin{bmatrix} W_{l;i} \\ W_{r;i} \end{bmatrix}$ should also be column orthonormal. In this way, we guarantee that the error does not accumulate through

its propagation. And we call a HSS representation with these properties a HSS representation in proper form. Note that: In the proper form, Bs do not have to be column orthonormal. B should be of low rank such that the HSS representation is efficient.

Lemma 1 *It is possible (nontrivially) to factor a block upper triangular matrix into the form:*

$$\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} = \begin{bmatrix} Q_1 & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & Q_2 \end{bmatrix} \tag{2.17}$$

Proof: First consider a QR factorization of the top row:

$$\begin{bmatrix} A_{11} & A_{12} \end{bmatrix} = Q_1 \begin{bmatrix} R_{11} & R_{12} \end{bmatrix} \tag{2.18}$$

followed by an LQ factorization of

$$\begin{bmatrix} R_{12} \\ A_{22} \end{bmatrix} = \begin{bmatrix} L_{12} \\ L_{22} \end{bmatrix} Q_2 \tag{2.19}$$

Then it follows that:

$$\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} = \begin{bmatrix} Q_1 & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} R_{11} & L_{12} \\ 0 & L_{22} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & Q_2 \end{bmatrix} \tag{2.20}$$

and our result follows with $B_{11} = R_{11}, B_{12} = L_{12}, B_{22} = L_{22}$.
QED.

With the left-split method at our deposal, we can split the root recursively until we are satisfied. The pseudo-code of this top-down method follows:

```
Given a plain matrix A and certain termination condition d.  Return the
HSS representation of A.

 1. Build a dummy leaf L with A, of which U = |, V = |, D = A.

 2. Node(node,left-leaf,right-leaf) = leaf-split L

 3. if (d = false) then
    return Node(node, leaf-split(left-leaf), leaf-split(right-leaf))

 4. else
    return Node(node, left-leaf, right-leaf)
```

Chapter 3

Hierarchically Semi-Separable Representation and Dataflow diagrams

In [37], a solver for HSS representation is presented. By cleverly converting a structured matrix in HSS representation into a larger sparse system of equations that has an ordering of the unknowns, a very efficient direct Gaussian elimination solver can be used. The solver is claimed to have linear time complexity. In [25], A efficient multi-way solver based on a tree data structure is given. In this chapter, we demonstrate Data processing diagram and some useful modifications on HSS dataflow with the aim to give a more intuitive demonstration on the fast HSS algorithms in [25] [24].

3.1 Dataflow diagram notations

In [25],[30] and [37], the HSS representation has been expressed as a binary tree based structure. Matrix operations in HSS form such as Matrix-Vector multiplication, solve, factorization and so on can be represented as traversing and editing this tree structure. In [16], signal processing diagram is adapted to represent these algorithms in a more intuitive way. However, because of the space limitation on the formal papers, those beautiful diagrams are not well explained. Thus we decide to illustrate these HSS algorithms introduced in [37] [25] [30] and [16] in dataflow diagrams. Specially, we shall demonstrate how the fast solver in [25] would work on the tree based structure. Before complicated dataflow diagrams are shown, we shall introduce the important sub-graphs, basic notations and elementary modifications of the dataflow. Then these diagrams will be used extensively to demonstrate our algorithms. In particular, the solver algorithm will be represented as a series of modifications and operations on the dataflow diagram of Matrix-Vector multiplication.

3.1.1 Dataflow notations

This subsection describes the basic elements and elementary operations which would form the dataflow diagrams. The vectors in these small circles represent intermediate states, pure inputs or pure outputs. the matrices on edges represent those translation matrices coming from Hss representation in [25].

Matrix Products

C is $M \times N$ matrix; D, $N \times P$; E, $M \times P$. $E = CD$.

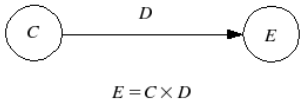

Figure 3.1: Matrix Products

Matrix Sums

F is $M \times N$ matrix; G, $P \times N$ matrix; x, y and H, N dimensional vectors, where $H = Fx + Gy$.

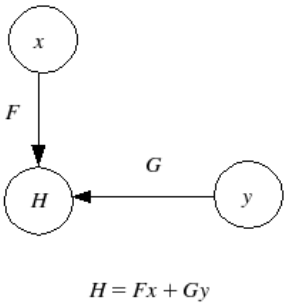

Figure 3.2: Matrix Sums

Matrix Split

Suppose that matrix A is of dimension $M \times N$. Then A_1 is of dimension $k \times N$; A_2, $(M-k) \times N$, where $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$.

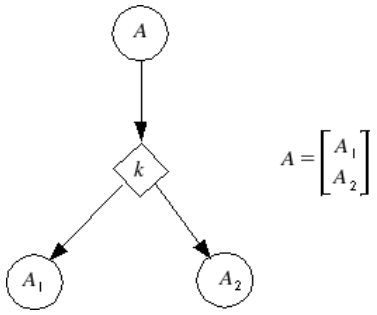

Figure 3.3: Matrix Split

Matrix Merge

Suppose A_1 is of dimension $M \times N$; A_2, $P \times N$; A, $(M+P) \times N$, where $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$.

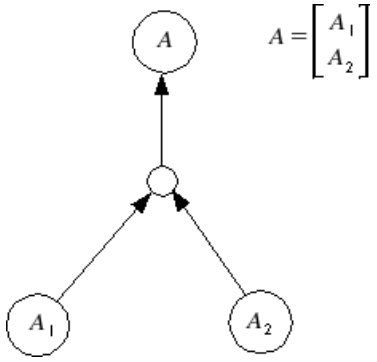

Figure 3.4: Matrix Merge

3.1.2 Elementary modifications on Dataflow

This subsection describes possible modifications which can be applied on the Dataflow diagrams. Some of them are quite obvious while the latter ones are more complicated and extensively used in solving and factorization algorithms. In this section, we just list them and prove their correctness. An example on how these modifications will be used to solve a linear system will be given at the end of this section.

Upward move of a factor

When we have multiple inputs while one output shown as in the following diagram, we can choose to factor the matrix to be multiplied on the output path and apply one of its factors to all inputs. In this way, the matrix on the output path could be simplified (only one of its factor will remain on the output path). The correctness of this modification is obvious.

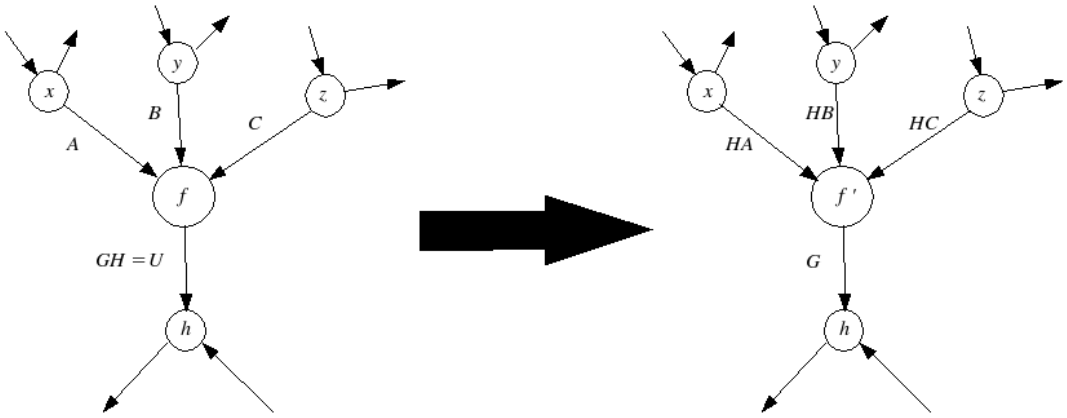

Figure 3.5: Upward move of a factor

Downward move of a factor

When we have multiple outputs while one input shown as in the following diagram, we can choose to factor the matrix to be multiplied on the input path and apply one of its factors to all outputs. In this way, the matrix on the input path could be simplified (only one of its factor will remain on the input path). The correctness of this modification is also obvious.

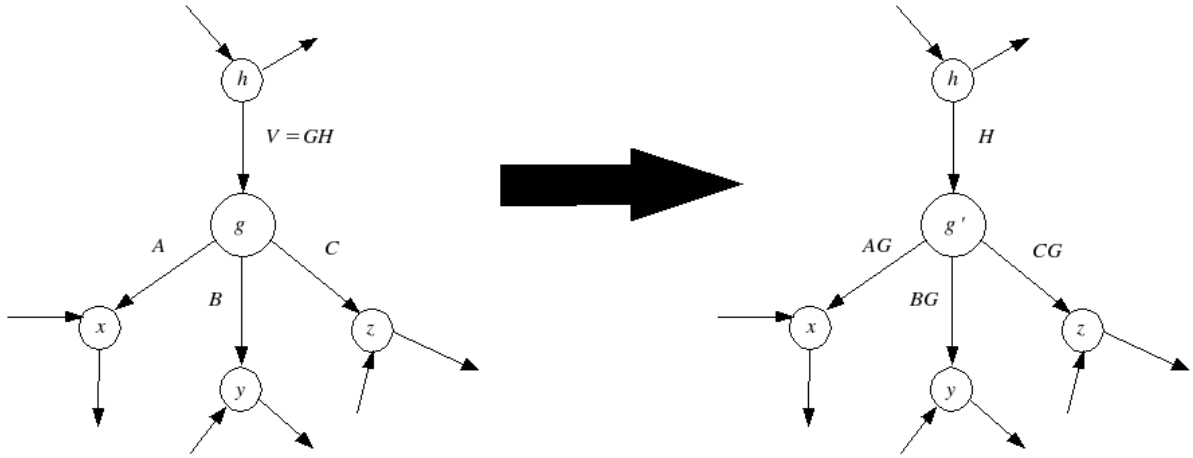

Figure 3.6: Downward move of factor

Downward propagation of a factor

Now we may have multiple inputs and multiple outputs, we can choose to factor the matrices to be multiplied on the input paths (note that the left factors of these input matrices should be the same) and apply one of the factors to all outputs. In this way, the matrices on the input paths could be simplified (only one of their factors will remain on the input path). The correctness of this modification is obvious.

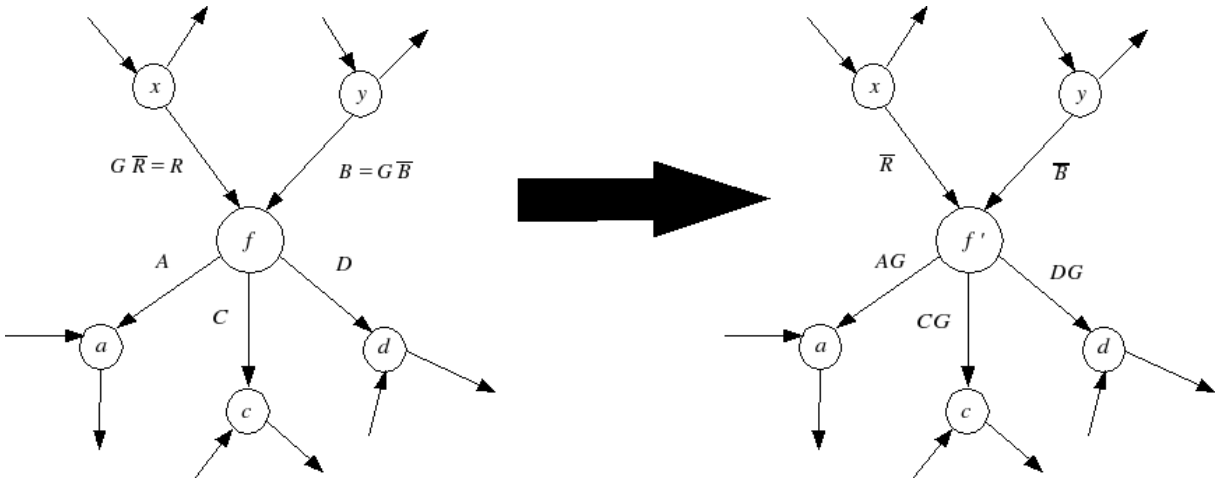

Figure 3.7: Downward propagation of a factor

Upward propagation of a factor

Alternatively, We can choose to factor the matrices to be multiplied on the output paths (note that the right factors of these output matrices should be the same) and apply one of the factors to all inputs. In this way, the matrices on the output paths could be simplified (only one of their factors will remain on the output path). The correctness of this modification is obvious.

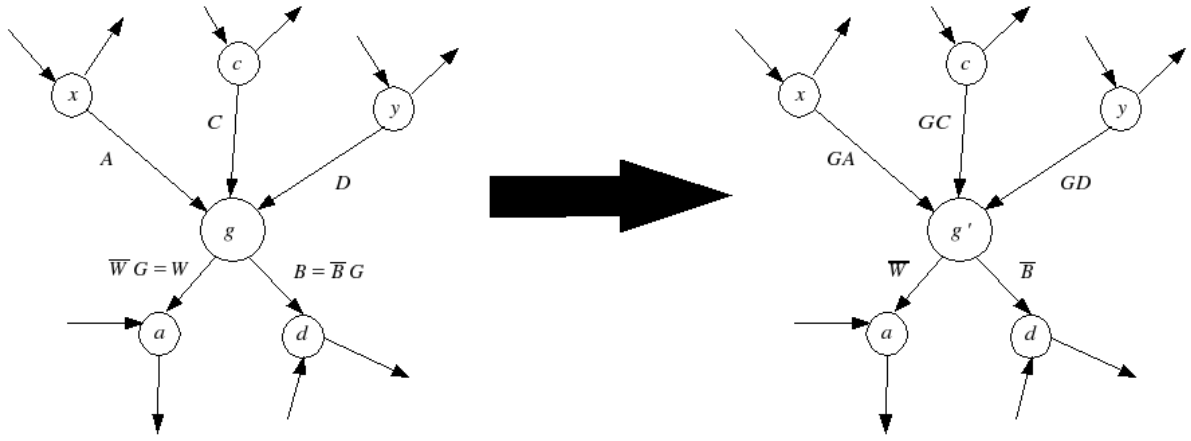

Figure 3.8: Upward propagation of a factor

Downward sliding of a link

When we have input which is relatively easy to compute, we can slide this input to the children on the output paths. However, in this way, the same computation may have to be done many times. Again, we can immediately see the correctness of this modification.

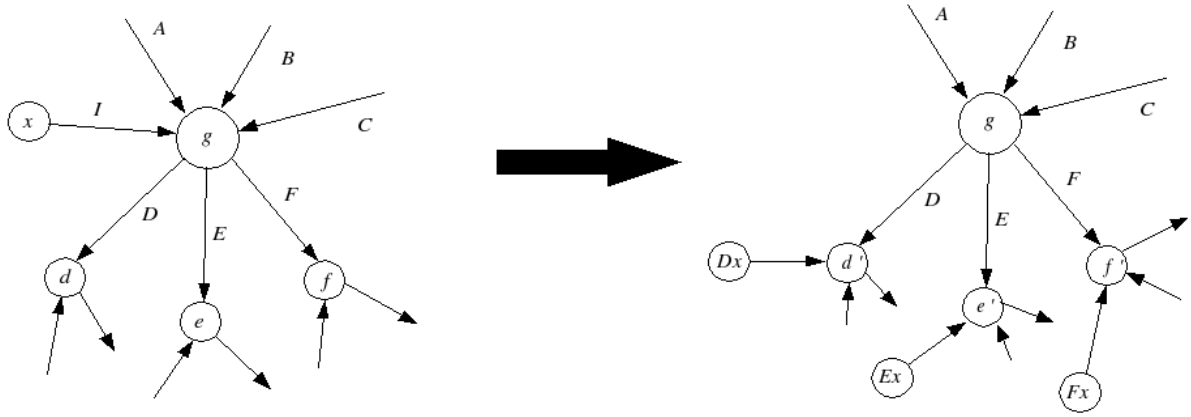

Figure 3.9: Downward sliding of a link

3.1.3 Special modifications

In the last subsection, we stated those simple modifications which are quite obvious. In this section, we will introduce more complicated modifications, which are quite related to the HSS representation. More precisely, they work on the dataflow diagram of the fast Matrix-Vector multiplication algorithm. we shall present the dataflow diagram of the fast matrix-vector multiplication algorithm of HSS in the coming paragraph.

In the diagram below, a multiplication dataflow diagram for 2-level HSS is presented. While for details on fast Matrix-Vector multiplication algorithm, refer to [4] and [5].

Actually, the dataflow diagram for fast Matrix-Vector multiplication($b = Ax$) is the most important dataflow diagram, because the fast solver[25] works on the same diagram, that is, given the multiplication dataflow diagram of $b = Ax$ (b, and A are given), the solver traverses and edits the dataflow diagram to find the unknown x. We give the data processing diagram below on fast multiplication algorithm of conventional HSS [1].

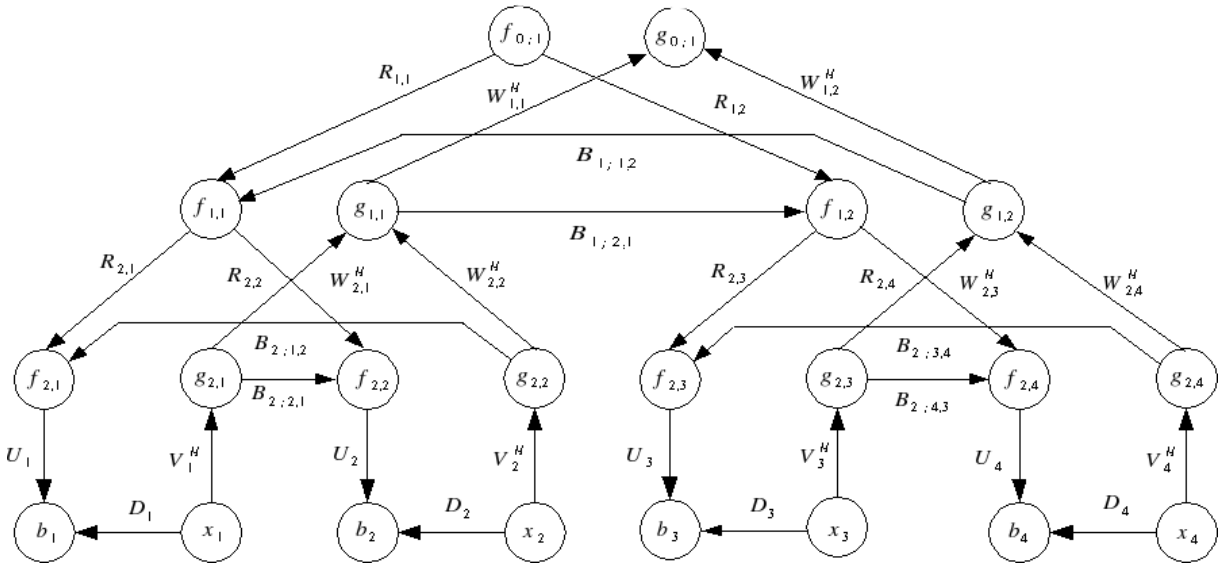

Figure 3.10: Multiplication of 2-level HSS (conventional)

To present this diagram in a comprehensive way, we shall explain its important sub-graphs in the coming paragraphs.

[1] we call it the conventional HSS, because we will introduce some of its variants later

Important sub-graphs

- Sub-graph for pure output b

  b is a proportion of matrix-vector multiplication output ($b = Ax$). Here, in this diagram, $b = Uf + Dx$, where f is an intermediate variable defined in equation 1.13.

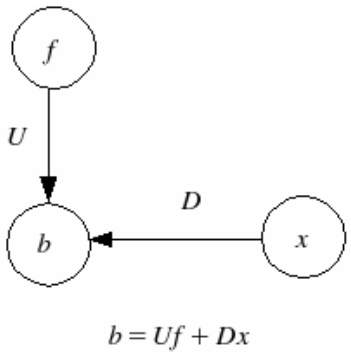

Figure 3.11: Sub-graph for b

- Sub-graph for intermediate g

  g is an important intermediate variable in Matrix-vector multiplication for HSS (defined in equation 1.13). It is assembled from the leaves and propagated through the nodes.

  When at the leaf of the HSS tree, $g = V^T x$.

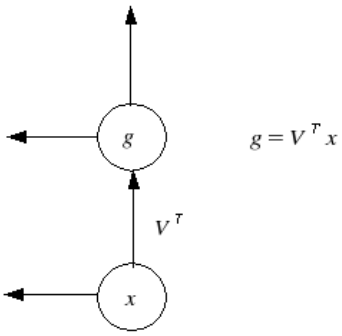

Figure 3.12: g at leaf

When at the nodes of HSS tree, $g = W_l^T g_l + W_r^T g_r$.

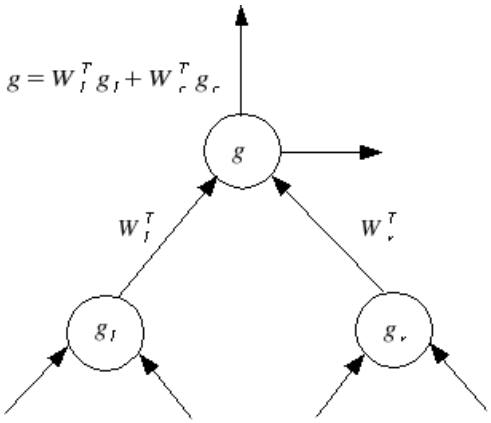

Figure 3.13: g at node

- Sub-graph for intermediate f

  f is another important intermediate variable in fast Matrix-vector multiplication for HSS (defined in equation 1.13). It is propagated from the root, through the nodes then to the leaves. Thus here we only have to give a formula to compute f at nodes. The f of the root is always initialized to be ϕ (dummy matrix defined in [5]).

  When at the node of the HSS tree, $f = Rf_p + Bg$.

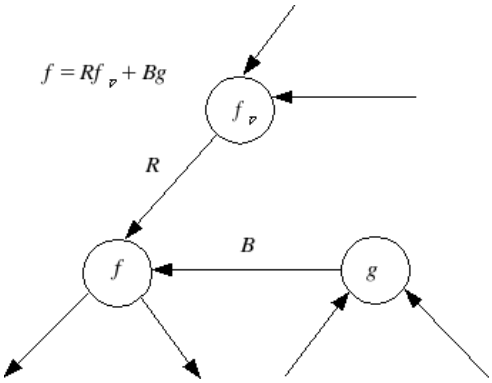

Figure 3.14: f at node

Leaf merge

Leaf merge is a special type of modification on the multiplication dataflow. Two leaves belonging to the same parent node can be merged to one single leaf, while the intermediate variables f and g of the parent node stay unchanged. It is actually the Merging operation in the Fast and Stable Adaptive Solver for Hierarchically Semi-separable Representations[5][33]. We shall give its Dataflow modification as follows, prove its correctness.

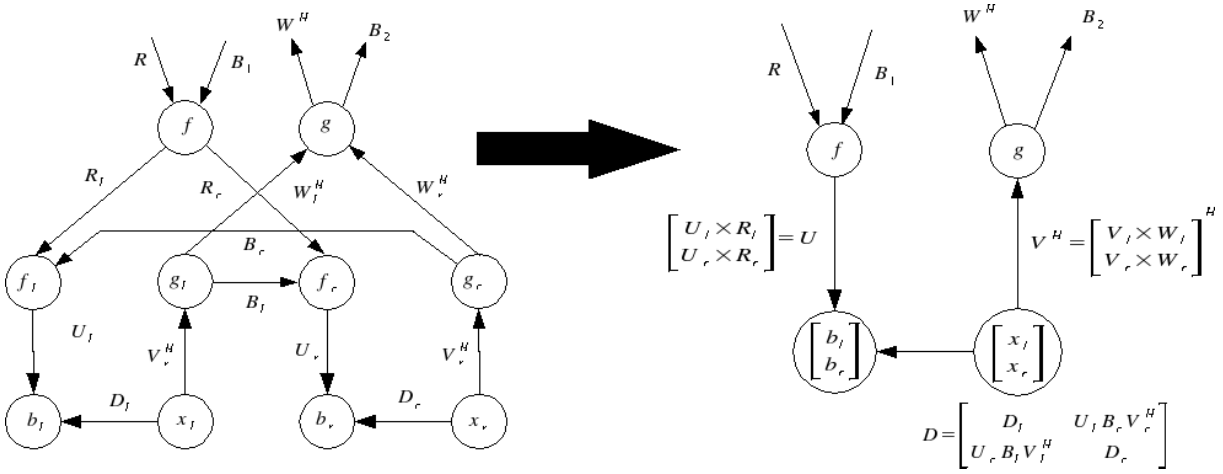

Figure 3.15: Leaf merge

Proof: From the left diagram, we know that:

$$g = V_l^H W_l^H x_l + V_r^H W_r^H x_r = \begin{bmatrix} V_l^H W_l^H & V_r^H W_r^H \end{bmatrix} \begin{bmatrix} x_l \\ x_r \end{bmatrix} = \begin{bmatrix} W_l V_l \\ W_r V_r \end{bmatrix}^H \begin{bmatrix} x_l \\ x_r \end{bmatrix} \tag{3.1}$$

While from the right one, we have:

$$g = V^H \begin{bmatrix} x_l \\ x_r \end{bmatrix} \tag{3.2}$$

So:

$$V = \begin{bmatrix} W_l V_l \\ W_r V_r \end{bmatrix} \tag{3.3}$$

From the left diagram, we have:

$$b_l = U_l f_l + D_l x_l \tag{3.4}$$

$$b_r = U_r f_r + D_r x_r \tag{3.5}$$

$$f_l = R_l f + B_r g_r \tag{3.6}$$

$$f_r = R_r f + B_r g_l \tag{3.7}$$

$$g_l = V_l^H x_l \tag{3.8}$$

$$g_r = V_r^H x_r \tag{3.9}$$

From the above 6 formulas, we derive the equations:

$$b_l = U_l R_l f + D_l x_l + U_l B_r V_r^H x_r \tag{3.10}$$

$$b_r = U_r R_r f + D_r x_r + U_r B_l V_l^H x_l \tag{3.11}$$

assemble the above two formulas as follows:

$$\begin{bmatrix} b_l \\ b_r \end{bmatrix} = \begin{bmatrix} U_l R_l \\ U_r R_r \end{bmatrix} f + \begin{bmatrix} D_l & U_l B_r V_r^H \\ U_r B_l V_l^H & D_r \end{bmatrix} \begin{bmatrix} x_l \\ x_r \end{bmatrix} \tag{3.12}$$

Let:

$$U = \begin{bmatrix} U_l R_l \\ U_r R_r \end{bmatrix} \tag{3.13}$$

$$D = \begin{bmatrix} D_l & U_l B_r V_r^H \\ U_r B_l V_l^H & D_r \end{bmatrix} \tag{3.14}$$

We have:

$$\begin{bmatrix} b_l \\ b_r \end{bmatrix} = Uf + D \begin{bmatrix} x_l \\ x_r \end{bmatrix} \tag{3.15}$$

We can assemble b_l and b_r, x_l and x_r together.Then we will get the exact dataflow on the right hand side. Q.E.D.

Leaf split

Leaf split is the reverse of leaf merge. One big leaf could be split into two smaller leaves, while the intermediate variables f and g of the big leaf will become the intermediate variables of the parent of the two new leaves. Leaf split modification can be done on any leaf. Particularly, it can be done on the root node of a one-level HSS, after which we can split as much as we needed. This may lead to a HSS construction algorithm in a top-down fashion (Note that, a construction is this fashion will not be space efficient $O(N^2)$, since we have to keep the original matrix.). It also serves as a way to prove the fact that every matrix A has its HSS representations. We give its Dataflow modification as below.

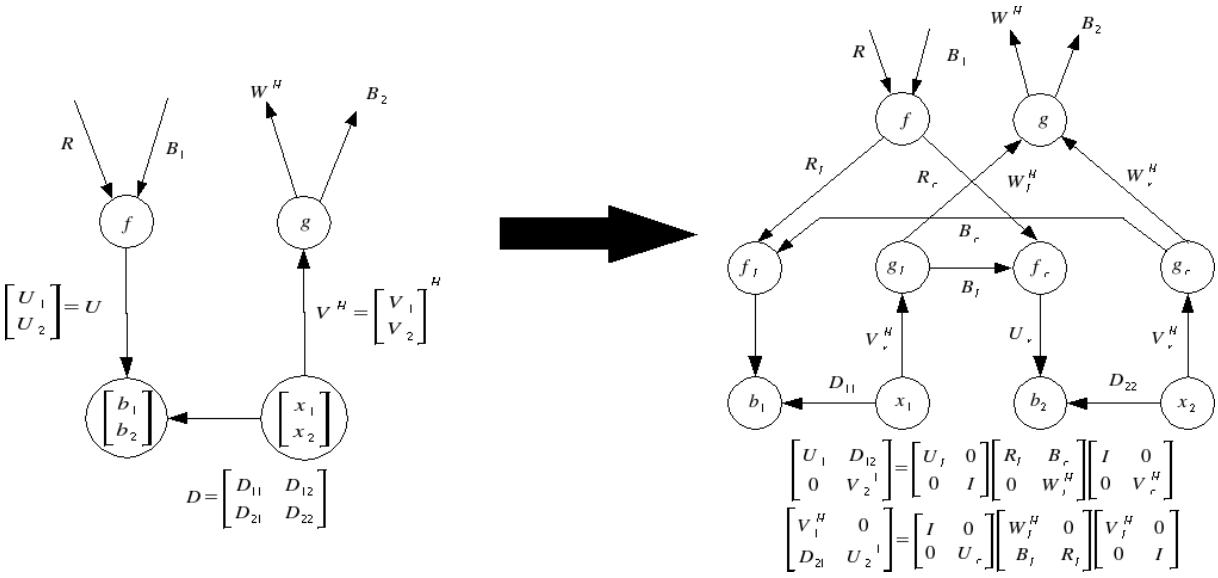

Figure 3.16: Leaf split

The proof of its correctness is omitted, as we can easily derive it from the proof of leaf merge. What has to be mentioned is the formula to generate the matrices of the new leaves from those of the big one. Firstly, we partition the matrices in the following way. Note that the dimensions of the partitions should be appropriate, so that the following calculations below are legal. Let:

$$U = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix}, V^H = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}^H, b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, D = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} \tag{3.16}$$

Then, the matrices in the right hand dataflow diagram should hold the following equations:

$$\begin{bmatrix} U_1 & D_{12} \\ 0 & V_2^H \end{bmatrix} = \begin{bmatrix} U_l & 0 \\ 0 & I \end{bmatrix} \begin{bmatrix} R_l & B_r \\ 0 & W_r^H \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & V_r^H \end{bmatrix} = \begin{bmatrix} U_l R_l & U_l B_r V_r^H \\ 0 & W_r^H V_r^H \end{bmatrix} \tag{3.17}$$

$$\begin{bmatrix} V_1^H & 0 \\ D_{21} & U_2 \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & U_r \end{bmatrix} \begin{bmatrix} W_l^H & 0 \\ B_l & R_r \end{bmatrix} \begin{bmatrix} V_l^H & 0 \\ 0 & I \end{bmatrix} = \begin{bmatrix} W_l^H V_l^H & 0 \\ U_r B_l V_l^H & U_r R_r \end{bmatrix} \tag{3.18}$$

Note that: U_i, V_i should be chosen such that they are column orthonormal. $\begin{bmatrix} R_{l;i} \\ R_{r;i} \end{bmatrix}$ and $\begin{bmatrix} W_{l;i} \\ W_{r;i} \end{bmatrix}$ should also be column orthonormal. In this way, we guarantee that the error does not accumulate through its propagation. And we call a HSS with these properties a HSS in proper form. Note that: In the proper form, B factors do not have to be column orthonormal. B should be of low rank such that the HSS representation is efficient.

Orthonormalize moves

As we mentioned before, U_i, V_i should be chosen such that they are column orthonormal. $\begin{bmatrix} R_{l;i} \\ R_{r;i} \end{bmatrix}$ and $\begin{bmatrix} W_{l;i} \\ W_{r;i} \end{bmatrix}$ should also be column orthonormal, such that the error does not accumulate through propagation. When this proper form is not guaranteed at the beginning, we could modify the HSS tree and make it into this proper form. Series of modifications on leaves and on nodes would be needed. To guarantee that one modification will not disturb the modifications before, we can modify the HSS tree in a bottom-up fashion. In this order, the partial proper form gained by one modification will not be canceled by other modifications. Elementary modifications could be done on leaves or on nodes as follows:

- **At leaf**

  When U_i, V_i are not column orthonormal, we can factorize U_i, V_i into $U_Q R_Q$, $V_Q W_Q$, where U_Q and V_Q are column orthonormal. Then we can sweep the factors R_Q and W_Q to the parent node by the elementary modifications we mentioned in subsection 3.1.2.

  Let:

  $$U = U_Q R_Q, V = V_Q W_Q \tag{3.19}$$

  where U_Q, V_Q are column orthonormal. Then we can modify the leaf as follows:

  Its correctness is guaranteed, since this modification is a combination of elementary modifications which have been proven.

- **At node**

  When the **translation** matrices[5, 6, 4] on the node are not in a proper form, we can factor them into proper form.

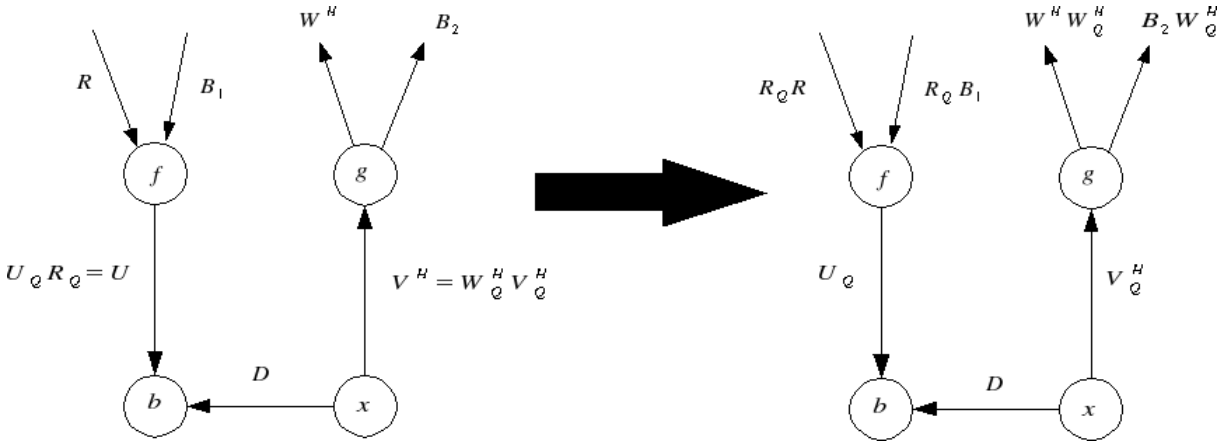

Figure 3.17: Orthonormal move at leaf

Let:

$$\begin{bmatrix} R_l \\ R_r \end{bmatrix} = \begin{bmatrix} Q_l \\ Q_r \end{bmatrix} \hat{R}, \begin{bmatrix} W_l \\ W_r \end{bmatrix} = \begin{bmatrix} P_l \\ P_r \end{bmatrix} \hat{W} \tag{3.20}$$

Here $\begin{bmatrix} Q_l \\ Q_r \end{bmatrix}$ and $\begin{bmatrix} P_l \\ P_r \end{bmatrix}$ are column orthonormal. Similar to the way we modify the leaf, we could move the factors to its parent. The diagram below shows how this is done on a node of HSS tree.

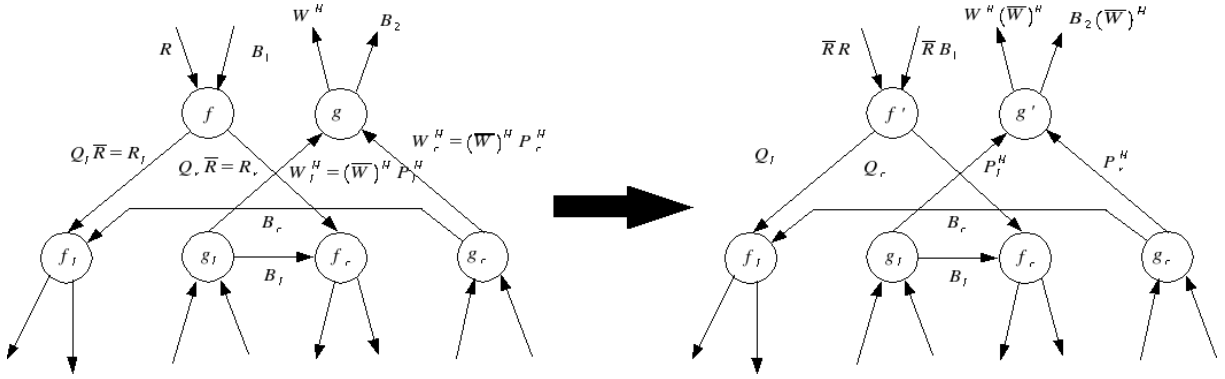

Figure 3.18: Orthonormal move at node

Again, since it is a combination of elementary modifications, its correctness is guaranteed given the factorization is correct.

We could apply the modifications above in a bottom-up fashion as many times as needed until we reach the root node, and by definition, the **translation** matrices in the root node are initialized to have no column at all. Thus, the factor propagated to the translation matrices of the root will be eliminated. Then the whole HSS tree will be in a proper form.

Note that: the translation matrices of the root node have no column at all, and one exception is made for them. They do not have to be column orthonormal.

Model reduction

There are some cases when certain error can be tolerated in particular to improve computation time. In these cases, we can do model reduction on the HSS tree to achieve speed-up at the cost of lose in accuracy. However a proper form of the HSS tree will be assumed such that the error incurred by model reduction will not be accumulated through propagation. Again, in order to keep the proper form intact, the modifications should be done in a top-down fashion. Two kinds of modification could be done to do the model reduction. They are:

- **Reduction at the root**

  If needed, we can start the model reduction from the root. We could factor T as $U_T \Sigma_T V_T^H + O(\epsilon)$, where U_T and V_T are column orthonormal. We could then sweep U_T and V_T to its children like this:

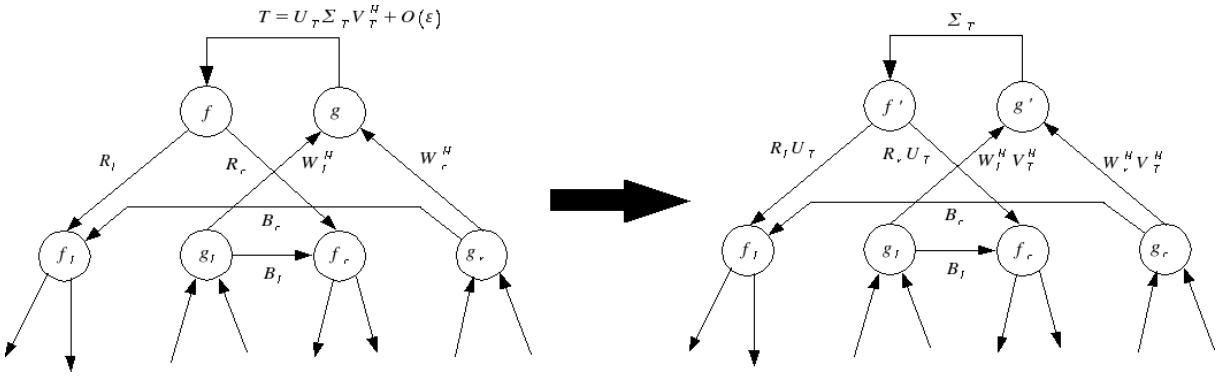

Figure 3.19: Model reduction at the root

  Note that, after the model reduction on the root, the translation matrices have become $\hat{R}_l = R_l U_T$, $\hat{R}_r = R_r U_T$, $\hat{W}_l = W_l V_T$, and $\hat{W}_r = W_r V_T$. then:

$$\begin{bmatrix} \hat{R}_l \\ \hat{R}_r \end{bmatrix} = \begin{bmatrix} R_l U_T \\ R_r U_T \end{bmatrix} = \begin{bmatrix} R_l \\ R_r \end{bmatrix} U_T \tag{3.21}$$

$$\begin{bmatrix} \hat{W}_l \\ \hat{W}_r \end{bmatrix} = \begin{bmatrix} W_l V_T \\ W_r V_T \end{bmatrix} = \begin{bmatrix} W_l \\ W_r \end{bmatrix} V_T \tag{3.22}$$

  If the given HSS tree is in a proper form, then $\begin{bmatrix} R_l \\ R_r \end{bmatrix}$ and $\begin{bmatrix} W_l \\ W_r \end{bmatrix}$ are column orthonormal. From the factorization we get U_T and V_T, which are column orthonormal, then $\begin{bmatrix} R_l \\ R_r \end{bmatrix} U_T$ and $\begin{bmatrix} W_l \\ W_r \end{bmatrix} V_T$ are column orthonormal. The modified HSS tree is again in a proper form.

- **Reduction at node/leaf**

  When needed, model reduction could also be done on nodes. Given a node like the one shown on the left of the diagram below, we can factor the matrices as follows:

Let:

$$\left[\begin{array}{cc} R_l & B_r \end{array}\right] = U_l \left[\begin{array}{cc} \hat{R}_l & \hat{B}_r \end{array}\right] + O(\epsilon) \tag{3.23}$$

$$\left[\begin{array}{cc} R_r & B_l \end{array}\right] = U_r \left[\begin{array}{cc} \hat{R}_r & \hat{B}_l \end{array}\right] + O(\epsilon) \tag{3.24}$$

$$\left[\begin{array}{c} W_r^H \\ \hat{B}_r \end{array}\right] = \left[\begin{array}{c} \hat{W}_r^H \\ B_r \end{array}\right] V_r^H + O(\epsilon') \tag{3.25}$$

$$\left[\begin{array}{c} W_l^H \\ \hat{B}_l \end{array}\right] = \left[\begin{array}{c} \hat{W}_l^H \\ B_l \end{array}\right] V_l^H + O(\epsilon') \tag{3.26}$$

Or equivalently:

$$\left[\begin{array}{cc} R_l & B_r \\ 0 & W_r^H \end{array}\right] = \left[\begin{array}{cc} U_l & 0 \\ 0 & I \end{array}\right] \left[\begin{array}{cc} \hat{R}_l & \hat{B}_r \\ 0 & W_r^H \end{array}\right] + O(\epsilon) = \left[\begin{array}{cc} U_l & 0 \\ 0 & I \end{array}\right] \left[\begin{array}{cc} \hat{R}_l & \tilde{B}_r \\ 0 & \hat{W}_r^H \end{array}\right] \left[\begin{array}{cc} I & 0 \\ 0 & V_r^H \end{array}\right] + O(\epsilon') \tag{3.27}$$

$$\left[\begin{array}{cc} R_r & B_l \\ 0 & W_l^H \end{array}\right] = \left[\begin{array}{cc} U_r & 0 \\ 0 & I \end{array}\right] \left[\begin{array}{cc} \hat{R}_r & \hat{B}_l \\ 0 & W_l^H \end{array}\right] + O(\epsilon) = \left[\begin{array}{cc} U_r & 0 \\ 0 & I \end{array}\right] \left[\begin{array}{cc} \hat{R}_r & \tilde{B}_l \\ 0 & \hat{W}_l^H \end{array}\right] \left[\begin{array}{cc} I & 0 \\ 0 & V_l^H \end{array}\right] + O(\epsilon') \tag{3.28}$$

Given the error can be tolerated, we can omit the error factors and compress the above equations in the following way.

$$\left[\begin{array}{cc} R_l & B_r \\ 0 & W_r^H \end{array}\right] \approx \left[\begin{array}{cc} U_l & 0 \\ 0 & I \end{array}\right] \left[\begin{array}{cc} \hat{R}_l & \tilde{B}_r \\ 0 & \hat{W}_r^H \end{array}\right] \left[\begin{array}{cc} I & 0 \\ 0 & V_r^H \end{array}\right] \tag{3.29}$$

$$\left[\begin{array}{cc} R_r & B_l \\ 0 & W_l^H \end{array}\right] \approx \left[\begin{array}{cc} U_r & 0 \\ 0 & I \end{array}\right] \left[\begin{array}{cc} \hat{R}_r & \tilde{B}_l \\ 0 & \hat{W}_l^H \end{array}\right] \left[\begin{array}{cc} I & 0 \\ 0 & V_l^H \end{array}\right] \tag{3.30}$$

Or equivalently:

$$R_l \approx U_l \hat{R}_l, R_r \approx U_r \hat{R}_r \tag{3.31}$$

$$W_l \approx V_l \hat{W}_l, W_r \approx V_r \hat{W}_r \tag{3.32}$$

$$B_r \approx U_l \tilde{B}_r V_r^H, B_l \approx U_r \tilde{B}_l V_l^H \tag{3.33}$$

Then we could sweep these factors to its children as follows:

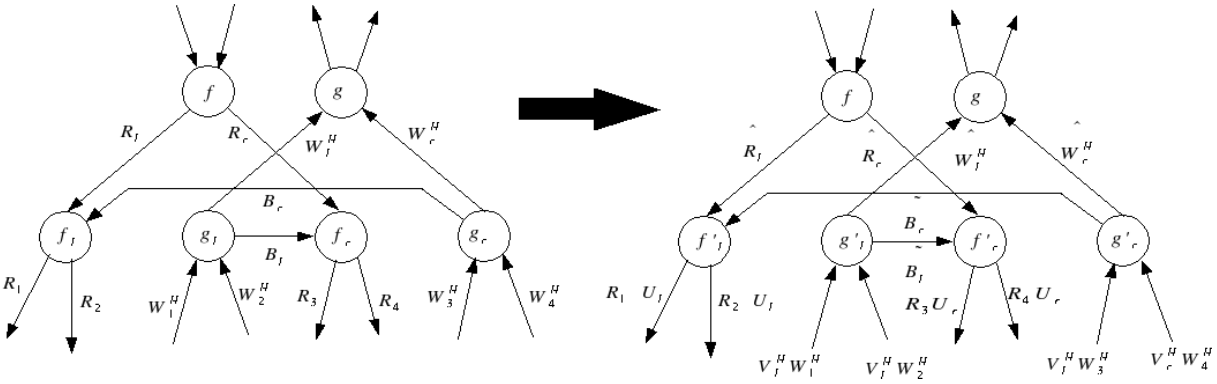

Figure 3.20: Model reduction at node

In order to give an idea on how this fast model reduction could reduce the complexity, we illustrate the HSS trees before model reduction and after model reduction as follows:

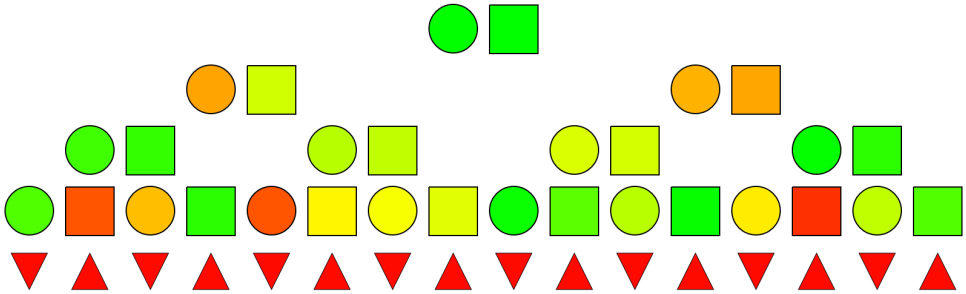

Figure 3.21: HSS tree before Model Reduction

The one above is a graphic representation of the HSS tree, while the small circles represents intermediate states f, the small rectangle represents intermediate states g, The small triangles represents pure inputs and pure outputs. The color of these entities represents the rank of the states, red means a higher rank while green indicates a lower one. Now after we specify a certain tolerance and do a model reduction on this HSS tree, we get the following reduced HSS tree:

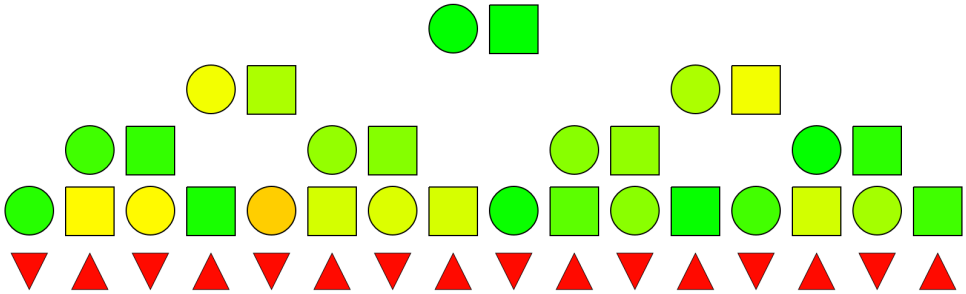

Figure 3.22: HSS tree after Model Reduction

We can see from the diagrams that the model reduction method has certainly reduced the dimension of the states, but how much it reduces the complexity really depends on the HSS tree (the rank of the translation matrices) and the tolerance specified.

3.1.4 Fast Solve

Finally, we are ready to represent the Fast and Stable Adaptive Solver algorithm in [5] as modifications and traverse on dataflow diagram of matrix-vector multiplication in HSS. As we mentioned, the Fast and Stable Adaptive Solver for HSS can be interpreted as modifying and traversing the dataflow diagram. Series of two main modifications will be done on the dataflow diagram, one is **Merge** which is Leaf Merging modification we mentioned before, the other one is **Compression** which is to reduce the complexity of the dataflow and compute partial results. These two methods will be used iteratively on the dataflow until the dataflow is simple enough to be solved in a direct way. Then a second traverse will be needed to recover the solutions from the partial results. In this subsection, we will illustrate Compression and Merge as a number of elementary modifications on leaves. Then we will explain how the two methods lead to a multi-way solver.

Compression on leaf

Suppose Compression is done on a leaf as shown below. We assume that the dimensions of U satisfy the requirement for Compression, that is U has more rows than columns.

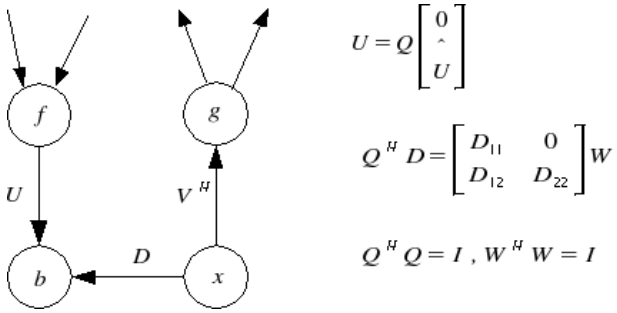

Figure 3.23: Compression(1)

Then we can factor the matrices according to the Compression algorithm in [5] as follows:

$$U = Q \begin{bmatrix} 0 \\ \hat{U} \end{bmatrix}, Q^H D = \begin{bmatrix} D_{11} & 0 \\ D_{12} & D_{22} \end{bmatrix} W \tag{3.34}$$

Here $Q^H Q = I$, $W^H W = I$, that is Q,W are column orthonormal. Then we alter the b and x with Q, W factors as follows:

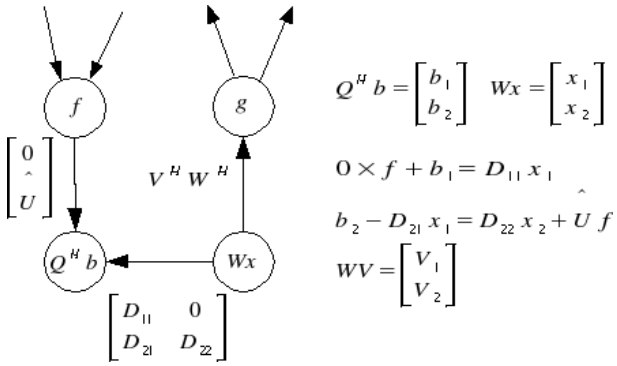

Figure 3.24: Compression(2)

Now, we apply Leaf Split modification on this altered leaf, then we get the node below. Note that: b should be altered before we do the operation below using the formula $b = b' - Uf$. b is coming from the upper level.

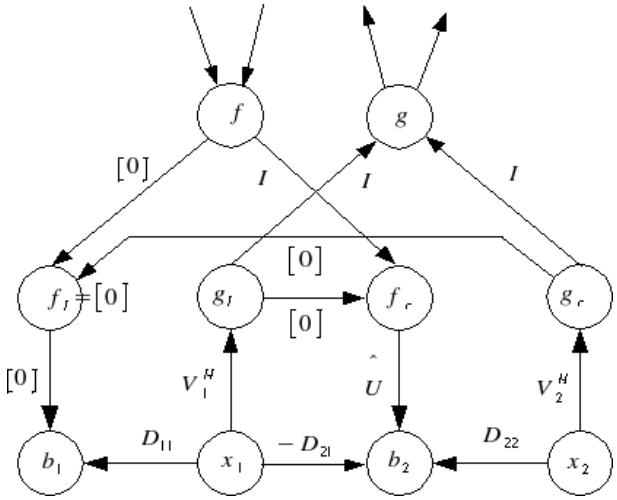

Figure 3.25: Compression(3)

Now, the edges with 0 matrices could be removed, since they contribute nothing to the dataflow. After this, the isolated intermediate nodes could be removed, since they are actually not needed. Then we have pruned the leaf into a concise one shown below

Again, since two states linked with identity matrix are identical, there is not need to keep both of them. We can merge these identical states. At the meantime, we calculate $-D_{21}x_1$ and $V_1^H x_1$. We then

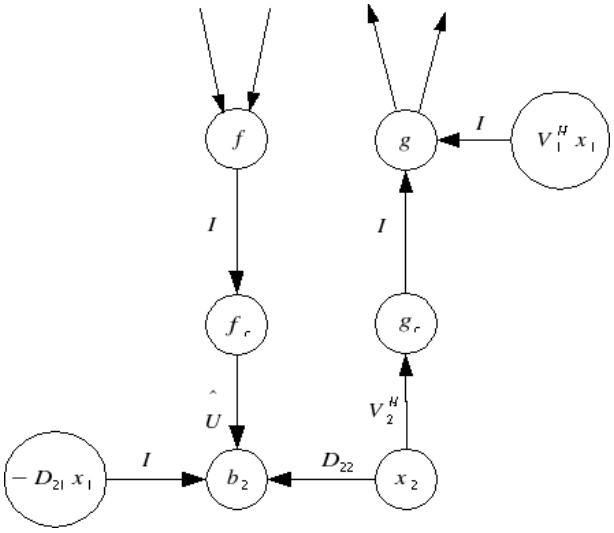

Figure 3.26: Compression(4)

modify b_2 as $b2' = b_2 - D_{21}x_1 - \hat{U}f$ and update g as $g' = V_1^H x_1$. (Note that: both f and g are initialized as 0s) After these two more vertexes can be omitted. The leaf is further simplified as follows:

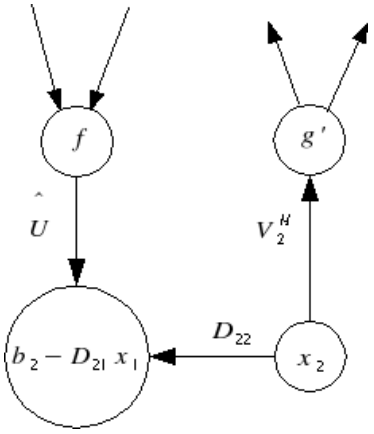

Figure 3.27: Compression(5)

After one Compression, we get a partial result x_1 (note that: this x_1 has been altered. It is not the same as the solution of the system.) and a smaller tree. let's first assume that: with the fast solver algorithm, we can get x_2 then we can recover the solution(x) of the system using the following formula:

$$Wx = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, x = W^{-1} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{3.35}$$

Because W is column orthonormal, we can recover x easily as $x = W^H \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

No Compression on leaf

If at a leaf U is not compressible, then there is nothing to do. The recursive algorithm will provide the solution at this leaf.

Merge on node

The Compression method we presented above is actually a method to reduce the complexity of the linear system. However it requires matrix U to have more rows than columns, while after a compression on a leaf, the number of rows will less or equal to the number of columns of the matrix U. When no compression can be done, the Merge method is needed to provide compressible leaves. Suppose, we have a node like the one below:

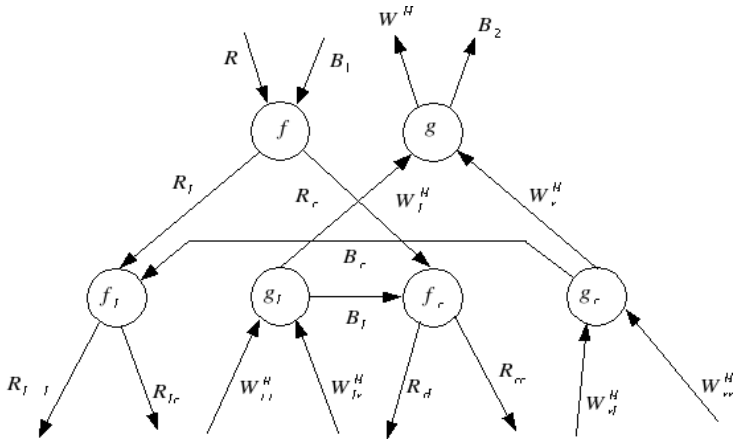

Figure 3.28: Merge(1)

First, we need to traverse the leaf children, with the intermediate variable $f_l = R_l f$. recursively using the algorithm we present. After its left children has been traversed, the left children will be merged as one leaf with its intermediate variable g_l calculated recursively.

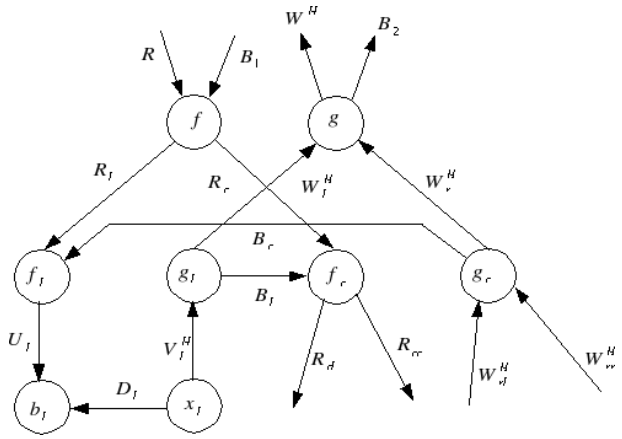

Figure 3.29: Merge(2)

Since the intermediate variable of the left leaf is known, we could then propagate g_l to the right children. That is we update the intermediate variables on its right leaf using the following formulas:

$$f_r = R_r f + B_l g_l \tag{3.36}$$

Then we traverse the right leaf according to the algorithm we present. The recursive algorithm will merge the right children into a leaf. After that we have a node with two leaves:

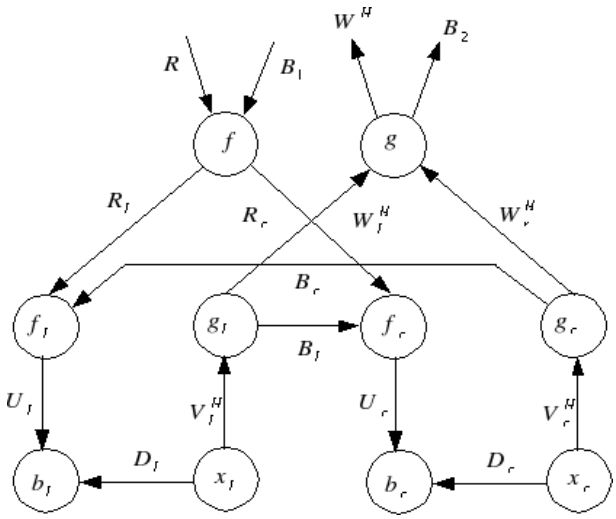

Figure 3.30: Merge(3)

After traversing the right leaf, we get the intermediate variable g_r from the right children, we should then update the intermediate variable g in the node using the following formula:

$$g = W_l^H g_l + W_r g_r \tag{3.37}$$

After all the intermediate variables of the node have been calculated, we can merge the two leaves and get a bigger leaf , The HSS representation at this leaf is given by the following formulas:

$$D = \begin{bmatrix} D_l & U_l B_r V_r^H \\ U_r B_l V_l^H & D_r \end{bmatrix} \tag{3.38}$$

$$U = \begin{bmatrix} U_l R_l \\ U_r R_r \end{bmatrix} \tag{3.39}$$

$$V = \begin{bmatrix} V_l W_l \\ V_r W_r \end{bmatrix} \tag{3.40}$$

Now, we get a new leaf in figure 3.31

Then, the system looks identical to the original system, except that it is guaranteed to have two fewer nodes at least. Suppose that the whole algorithm starts from the root node, we could end up with a single leaf with less unknowns. Then the system is small enough to be solved directly. Once this has been done,

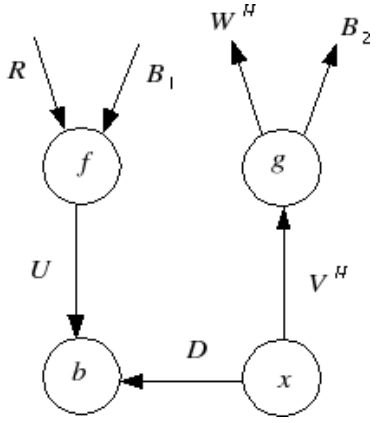

Figure 3.31: Merge(4)

the information in x' (Note that: the x' has been altered) must be passed along to the children. So for each node we partition the x as:

$$x = \begin{bmatrix} x_l \\ x_r \end{bmatrix} \tag{3.41}$$

x_l will be passed to the left children, while x_r will be passed to the right children. A second top-down traverse will then begin to recover the original solution.

After the second top-down traverse finishes, the original solution can be assembled from the bottom leaves.

3.2 Variant of HSS

We have proved that every matrix can be represented in HSS representation, however, this representation only suits those matrices which are of low rank on off diagonal blocks. When those off diagonal blocks are not of low rank, the multiplication, factorization, solver algorithm for this representation will be quite inefficient. In that case, the B,R and W matrices of the off diagonal blocks will not be as small as they are intended to be. In the worst case, the off diagonal matrices could be of full rank. Then the time complexity of these algorithms will be even worse than those direct methods on plain matrices. One possible solution is to swap the rows and columns of the matrix and make the off-diagonal blocks really low-rank, while that would perturb the whole system matrix and could be quite complicated.

Another possible solution is to use richer partitioning for the off-diagonal blocks. that is, we choose not only to partition on-diagonal matrices but also to partition off-diagonal matrices. Yet, we still want to keep the assumption that the off diagonal matrices have lower ranks than the on diagonal matrices. So the off diagonal matrices should not be partitioned as on-diagonal ones. The main idea is to partition off diagonal blocks as needed, that is to choose not to have a large B for the whole off diagonal block, but to have a few smaller Bs.while examples of variant for 4×4 blocks are given as follows:

$$\begin{bmatrix} D_{2,1} & U_{2,1}B_{2;1,2}V_{2,2}^H & U_{2,1}R_{2,1}B_{1;1,2(1)}W_{2,3}^HV_{2,3}^H & U_{2,1}R_{2,1}B_{1;1,2(2)}W_{2,4}^HV_{2,4}^H \\ U_{2,2}B_{2;2,1}V_{2,1}^H & D_{2,2} & U_{2,2}R_{2,2}B_{1;1,2(3)}W_{2,3}^HV_{2,3}^H & U_{2,2}R_{2,2}B_{1;1,2(4)}W_{2,4}^HV_{2,4}^H \\ U_{2,3}R_{2,3}B_{1;2,1(1))}W_{2,1}^HV_{2,1}^H & U_{2,3}R_{2,3}B_{1;2,1(2)}W_{2,2}^HV_{2,2}^H & D_{2,3} & U_{2,3}B_{2;3,4}V_{2,4}^H \\ U_{2,4}R_{2,4}B_{1;2,1(3)}W_{2,1}^HV_{2,1}^H & U_{2,4}R_{2,4}B_{1;2,1(4)}W_{2,2}^HV_{2,2}^H & U_{2,4}B_{2;4,3}V_{2,2}^H & D_{2,4} \end{bmatrix}$$

With these minor change on the HSS representation, the rank of Bs could be smaller. Yet the fast algorithms for it are not quite different from the ones for original HSS. To demonstrate how we should

modify the fast algorithms, we give data processing diagram of the fast multiplication in this representation is given below, yet similar changes can be made on other algorithms:

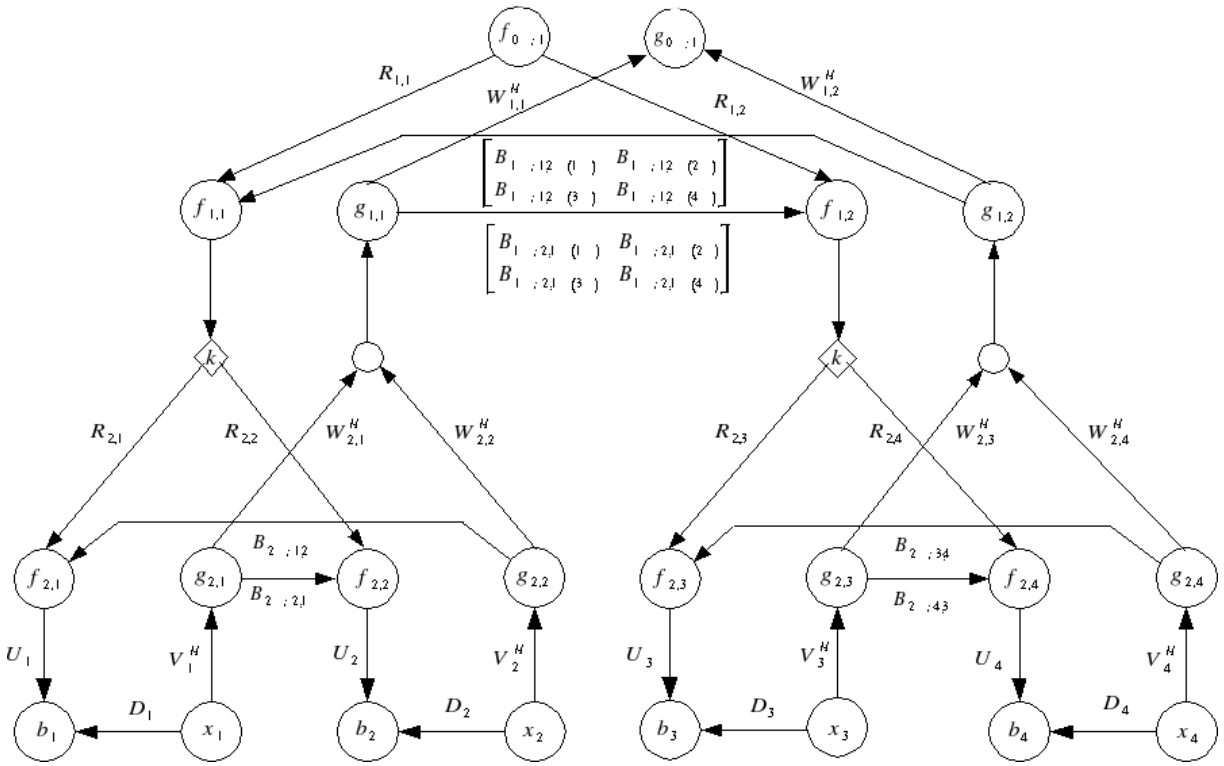

Figure 3.32: Multiplication of the first HSS variant

To push this idea further we can have more flexible partitions of the off diagonal blocks, that is to choose the same B for those sub-matrices which are far from the main diagonal, while choose different B for these sub-matrices which are nearer to the main diagonal. One can actually see it as a N-matrices fashion partitioning in [22], except here we have translation matrices. A possible example on 4×4 two-level HSS may be the following:

$$\begin{bmatrix} D_{2,1} & U_{2,1}B_{2;1,2}V_{2,2}^H & U_{2,1}R_{2,1}B_{1;1,2(1)}W_{2,3}^HV_{2,3}^H & U_{2,1}R_{2,1}B_{1;1,2(1)}W_{2,4}^HV_{2,4}^H \\ U_{2,2}B_{2;2,1}V_{2,1}^H & D_{2,2} & U_{2,2}R_{2,2}B_{1;1,2(2)}W_{2,3}^HV_{2,3}^H & U_{2,2}R_{2,2}B_{1;1,2(1)}W_{2,4}^HV_{2,4}^H \\ U_{2,3}R_{2,3}B_{1;2,1(1))}W_{2,1}^HV_{2,1}^H & U_{2,3}R_{2,3}B_{1;2,1(2)}W_{2,2}^HV_{2,2}^H & D_{2,3} & U_{2,3}B_{2;3,4}V_{2,4}^H \\ U_{2,4}R_{2,4}B_{1;2,1(1)}W_{2,1}^HV_{2,1}^H & U_{2,4}R_{2,4}B_{1;2,1(1)}W_{2,2}^HV_{2,2}^H & U_{2,4}B_{2;4,3}V_{2,2}^H & D_{2,4} \end{bmatrix}$$

This representation could also result in smaller translation matrices. And modifications have to be made on its algorithms. This representation becomes quite similar to the FMM representation, which has N-Matrix partitioning on the off-diagonal sub matrices.

3.3 Dataflow diagram of Fast multiplication in SSS

In fact, we can even view Sequential Semi-separable representation as one extreme case of these variants with sequential state equations. The data processing diagram is so convenient to use that we can represent the fast multiplication of SSS as follows:

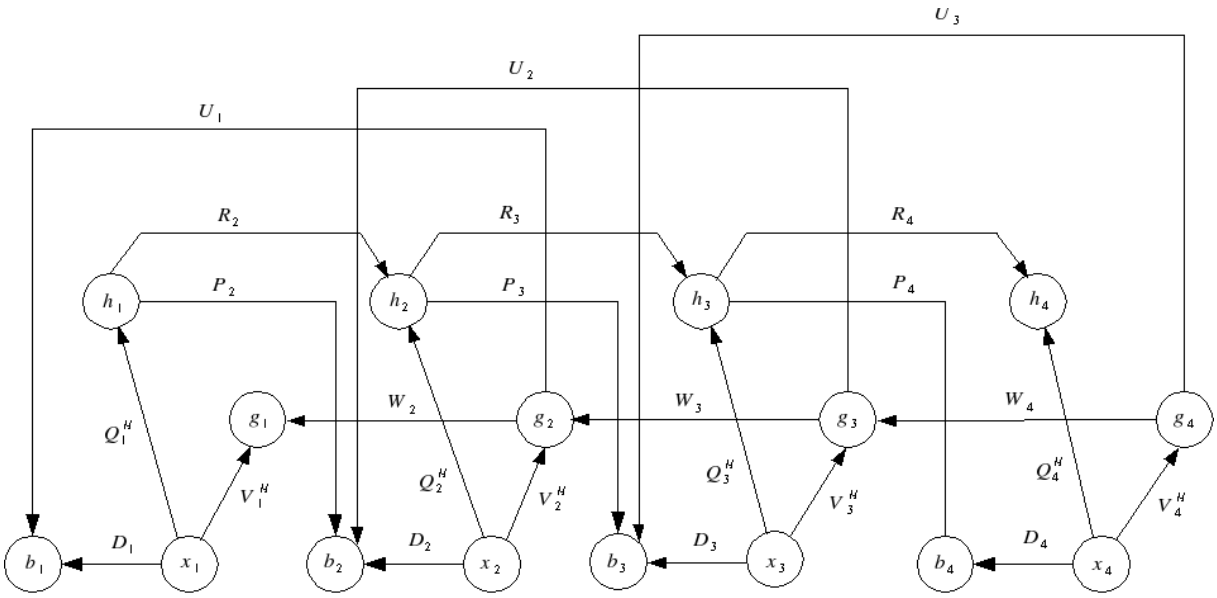

Figure 3.33: Multiplication of SSS (sequentially semi-separable structure)

Chapter 4

Algorithms for Hierarchically Semi-separable Representation

'Hierarchical Semi Separable' matrices (HSS matrices) form an important class of structured matrices for which matrix transformation algorithms that are linear in the number of equations (and a function of other structural parameters) can be given. In particular, a system of linear equations $Ax = b$ can be solved with linear complexity. Also, LU and URV factorization can be efficiently executed. There is a close connection between HSS matrices and classical Semi Separable matrices (sometimes called 'Sequentially Semi Separable matrices or SSS matrices or equivalently, time-varying systems), and a number of results of SSS theory can be translated to parallel results in the HSS theory (neither class contains the other however). This chapter gives a survey of the main results, including a complexity analysis and experimental results based on an OCAML implementation.

4.1 Matrix operations based on HSS representation

In this section we describe a number of basic matrix operations based on the HSS representation. Matrix operations using the HSS representation are normally much more efficient than operations with plain matrices. Many matrix operations can be done with a computational complexity (or sequential order of basic operations) linear with the dimension of the matrix. These fast algorithms to be described are either collected from other publications[4, 6, 23, 17],etc... or new. We will handle a somewhat informal notation to construct new block diagonals. Suppose e.g. that R_A and R_B are block diagonal matrices from the description given in the preceding section, then the construction operator inter$[R_A|R_B]$ will represent a diagonal operator in which the diagonal entries of the two constituents are block-column-wise intertwined:

$$\text{inter}[R_A|R_B] = \text{diag}[R_{A;1;1}|R_{B;1;1}, R_{A;1;2}|R_{B;1;2}, R_{A;2;1}|R_{B;2;1}, \cdots].$$

Block-row intertwining or matrix intertwining are defined likewise.

4.1.1 HSS addition

Matrix addition can be done efficiently with HSS representations. The addition algorithm for Sequentially semi-separable representation has been presented in [33]. The addition algorithm for HSS representation which has been studied in [23] is quite similar.

Addition with two commensurately partitioned HSS matrices

When adding two HSS commensurately partitioned matrices together, the sum will be an HSS matrix with the same partitioning. Let $C = A + B$ where A is defined by sequences U_A, V_A, D_A, R_A, W_A and B_A ; B is defined by sequences U_B, V_B, D_B, R_B, W_B and B_B.

- HSS addition at the node

$$\begin{cases} R_C = \text{inter} \left[\begin{array}{c|c} R_A & 0 \\ \hline 0 & R_B \end{array} \right] \\ W_C = \text{inter} \left[\begin{array}{c|c} W_A & 0 \\ \hline 0 & W_B \end{array} \right] \\ B_C = \text{inter} \left[\begin{array}{c|c} B_A & 0 \\ \hline 0 & B_B \end{array} \right] \end{cases} \tag{4.1}$$

- HSS addition at the leaf

$$U_C = \text{inter} \left[\begin{array}{c|c} U_A & U_B \end{array} \right] \quad V_C = \text{inter} \left[\begin{array}{c|c} V_A & V_B \end{array} \right] \quad D_C = D_A + D_B \tag{4.2}$$

The addition can be done in time proportional to the number of entries in the representation. Note that the computed representation of the sum may not be efficient, in the sense that the HSS complexity of the sum increases additively. It is quite possible that it is not minimal as well. In order to get an efficient HSS representation back, we could do *fast model reduction*(described in [17]) or even *compression*(to be presented later) on the resulting HSS representation. However, these operations might be too costly to be applied frequently, one could do *model reduction* or *compression* after a number of additions.

Adaptive HSS addition

When two HSS matrices of the same dimensions do not have the same depth, leaf-split or leaf-merge operations described in [17] are needed to make these two HSS representations compatible. Note that we have two choices: we can either split the shallower HSS tree to make it compatible with the deeper one, or we can do leaf-merge on the deeper tree to make it compatible with shallower one. From the view of computation complexity, leaf-merge is almost always preferred since it amounts to several matrix multiplications with small (ideally) matrices, while leaf-split needs several factorization operations which are more costly than matrix multiplications. However this does not imply leaf-merge should always be used if possible. Keeping in mind the fact that the efficiency of HSS representation also comes from a deeper HSS tree with smaller translation matrices, the HSS tree should be kept necessarily deep to capture the low rank off-diagonal blocks. On the other hand, it is obviously not possible to always do leaf-merge nor do leaf-split, because one HSS tree may have both a deeper branch and a shallower one than the other HSS tree does.

HSS addition with rank-k matrices

The sum of a hierarchically semi-separable matrix A and a rank-k matrix UBV^H is another hierarchically semi-separable matrix $A' = A + UBV^H$. Since a rank-k matrix is a hierarchically semi-separable matrix, we can represent the rank-k matrix UBV^H in HSS representation. Then the HSS addition described in the previous section can be performed.

- On the node level

  In order to add two matrices together, the rank-k matrix should be represented in the form compatible with the HSS matrix. That is, the rank-k matrix will have to be partitioned recursively according to the partition of the HSS matrix.

  We first partition U and V according the partition of matrix A as follows:

$$U = \begin{bmatrix} U_1 \\ U_2 \end{bmatrix} \quad V = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}$$

  to obtain

$$UBV^H = \begin{bmatrix} U_1 B V_1^H & U_1 B V_2^H \\ U_2 B V_1^H & U_2 B V_2^H \end{bmatrix}$$

  Then the HSS representation of the node becomes quite obvious:

$$\begin{cases} D_l = U_1 B V_1^H & D_r = U_2 B V_2^H & R_l = I & R_r = I \\ W_l = I & W_r = I & B_u = B & B_l = B \end{cases} \tag{4.3}$$

  Now D_l and D_r are again rank-k matrices, assuming the correctness of this constructive method, D_l and D_r can then be partitioned recursively.

- On the leaf level
  When enough partitions have been done, leaves will be constructed at the bottom level. We define the leaves as:

$$\hat{U} = U \quad D = UBV^H \quad \hat{V} = V \tag{4.4}$$

  where U and V are row bases and column bases coming from the its parent node.

Having the compatible HSS representation of the rank-k matrix,the HSS addition described in this section can be performed to compute the sum. Other ways of constructing HSS representation for rank-k matrix are also possible. One is to first form a one-level HSS representation for the rank-k matrix and then use leaf-split algorithm[17] to compute its HSS representation according to certain partitioning. In principle, this method leads to an efficient HSS tree in the sense that its column bases and row bases are irredundant. However, this method needs much more computations. If k is reasonably small, the method described in this section is recommended. Another point is that in practice the HSS representation of the rank-k should not be constructed explicitly to save on memory.

HSS addition with rank-k matrices with hierarchically semi-separable bases

In HSS representations, the column bases and row bases of the HSS nodes are not explicitly stored. This means when we compute $A' = A + UBV^H$, U and V are probably not explicitly stored, instead, they are implicitly stored with the formulas (1.10) and (1.11).

We can of course compute these row bases and column bases and then construct an HSS representation for UBV^H with the method described in the last sub-section. This is not recommended, because, computing U and V may be costly; and it is not memory efficient. Suppose U and V are defined in HSS form. The HSS representation of UBV^H can be constructed using the following top-down recursive formulas (4.5) to (4.8).

- At the top node:

$$\begin{cases} \hat{W}_l = I & \hat{W}_r = I & \hat{R}_l = I \\ \hat{R}_r = I & \hat{B}_l = B & \hat{B}_u = B \end{cases} \tag{4.5}$$

- At the middle nodes:

$$\begin{cases} \hat{W}_l = W & \hat{W}_r = W & \hat{R}_l = R \\ \hat{R}_r = R & \hat{B}_l = B & \hat{B}_u = B \end{cases} \tag{4.6}$$

  let

$$\tilde{B}_l = R_l B W_l^H, \tilde{B}_r = R_r B W_r^H \tag{4.7}$$

  Where B, W, R are coming from its parent node, $\tilde{B}_l$, W_l and R_l are propagated to its left child, and B_r, W_r and R_r are propagated to the right child.

- At the leaves

$$\hat{U} = \tilde{U} R \quad \hat{V} = \tilde{V} W \quad \hat{D} = \tilde{U} \tilde{B} \tilde{V}^H \tag{4.8}$$

  Where $\tilde{U}$ and $\tilde{V}$ are the row and column bases on the leaf level. These bases are explicitly stored. B, R and W matrices are coming from its parent.

After having the HSS representation of UBV^H, the sum can be computed easily using HSS addition algorithm described in section 4.1.1.

4.1.2 HSS matrix-matrix multiplication

Matrix-matrix multiplication can also be done in time linearly with the dimension of the matrices. The product $C = AB$ is another hierarchically semi-separable matrix.

A is a HSS matrix whose HSS representation is defined by sequences U_A, V_A, D_A, R_{Ar}, R_{Al}, W_{Ar}, W_{Al}, B_{Al} and B_{Au}.

B is a HSS matrix whose HSS representation is defined by sequences U_B, V_B, D_B, R_{Br}, R_{Bl}, W_{Br}, W_{Bl}, B_{Bl} and B_{Bu}.

Multiplication of two commensurately partitioned HSS matrices

When two HSS matrices are compatible, that is, they are commensurately partitioned, we can get the HSS representation of the product with the following algorithm.

The notations f and g to be used in following paragraphs represent the intermediate variables representing intermediate states in computing the HSS representation of C. They can be computed using the recursive formulas (4.9) to (4.12).

f_l represents f intermediate variable propagated to the left children; similarly, f_r represents the intermediate f propagated to the right children. g_l represents the intermediate variable g coming from the left children; while g_r represents the intermediate variable g coming from the right ones. At last, g_{leaf} represents the variable g calculated at leaves.

We first define the intermediate variables recursively via:

$$g_{\text{leaf}} = V_A^H U_B \tag{4.9}$$

$$g = W_{Al}^H g_l R_{Bl} + W_{Ar}^H g_r R_{Br} \tag{4.10}$$

$$f_l = B_{Al} g_r B_{Bu} + R_{Al} f W_{Bl}^H \tag{4.11}$$

$$f_r = B_{Au} g_l B_{Bl} + R_{Ar} f W_{Br}^H \tag{4.12}$$

Then the HSS representation of the product will be:

$$\begin{cases} D = D_A D_B + U_A f V_B^H & \\ U = \text{inter}\left[\begin{array}{c|c} U_A & D_A U_B \end{array}\right] & V = \text{inter}\left[\begin{array}{c|c} D_B^H V_A & V_B \end{array}\right] \\ R_l = \text{inter}\left[\begin{array}{c|c} R_{Al} & B_{Al} g_r R_{Br} \\ \hline 0 & R_{Bl} \end{array}\right] & R_r = \text{inter}\left[\begin{array}{c|c} R_{Ar} & B_{Au} g_l R_{Bl} \\ \hline 0 & R_{Br} \end{array}\right] \\ W_l = \text{inter}\left[\begin{array}{c|c} W_{Al} & 0 \\ \hline B_{Bu}^H g_r^H W_{Ar} & W_{Bl} \end{array}\right] & W_r = \text{inter}\left[\begin{array}{c|c} W_{Ar} & 0 \\ \hline B_{Bl}^H g_l^H W_{Al} & W_{Br} \end{array}\right] \\ B_l = \text{inter}\left[\begin{array}{c|c} B_{Al} & R_{Al} f W_{Br}^H \\ \hline 0 & B_{Bl} \end{array}\right] & B_u = \text{inter}\left[\begin{array}{c|c} B_{Au} & R_{Ar} f W_{Bl}^H \\ \hline 0 & B_{Bu} \end{array}\right] \end{cases} \tag{4.13}$$

Once again, the complexity of the HSS representation increases additively. *Model reduction* or *compression* may be needed to bring down the complexity. Note that, the algorithm above is given without proof. For a detailed proof and analysis, refer to [23].

Adaptive HSS Matrix-Matrix multiplication

Again, adaptive multiplication is needed when two HSS matrices are not completely compatible, then leaf-split and leaf-merge are needed to make them compatible. The same comment given in section (4.1.1) for adaptive addition also applies here.

HSS Matrix-Matrix multiplication with rank-k matrices

As we mentioned before, a rank-k matrix is a hierarchically semi-separable matrix and can be represented in a HSS representation. Then We can easily construct the HSS representation for the rank-k matrix and then perform HSS Matrix-Matrix multiplication. This is the most straightforward way, however, Making use of the fact the translation matrices (R,W) of the rank-k matrix are identity matrices, this algorithm can be improved a lot. Then formulas (4.9) to (4.13) can be written as (the same notations in section 4.1.2 are adapted):
We first define the intermediate variables recursively via:

$$g_{\text{leaf}} = V_A^H U_B$$

$$g = W_{Al}^H g_l + W_{Ar}^H g_r$$

$$f_l = B_{Al} g_r B_{Bu} + R_{Al} f$$

$$f_r = B_{Au} g_l B_{Bl} + R_{Ar} f$$

Then the HSS representation of the product will be simplified as:

$$\begin{cases} D = D_A D_B + U_A f V_B^H & \\ U = \text{inter}\left[\begin{array}{c|c} U_A & D_A U_B \end{array}\right] & V = \text{inter}\left[\begin{array}{c|c} D_B^H V_A & V_B \end{array}\right] \\ R_l = \text{inter}\left[\begin{array}{c|c} R_{Al} & B_{Al} g_r \\ \hline 0 & I \end{array}\right] & R_r = \text{inter}\left[\begin{array}{c|c} R_{Ar} & B_{Au} g_l \\ \hline 0 & I \end{array}\right] \\ W_l = \text{inter}\left[\begin{array}{c|c} W_{Al} & 0 \\ \hline B_{Bu}^H g_r^H W_{Ar} & I \end{array}\right] & W_r = \text{inter}\left[\begin{array}{c|c} W_{Ar} & 0 \\ \hline B_{Bl}^H g_l^H W_{Al} & I \end{array}\right] \\ B_l = \text{inter}\left[\begin{array}{c|c} B_{Al} & R_{Al} f \\ \hline 0 & B_{Bl} \end{array}\right] & B_u = \text{inter}\left[\begin{array}{c|c} B_{Au} & R_{Ar} f \\ \hline 0 & B_{Bu} \end{array}\right] \end{cases}$$

Because the complexity has been increased additively, *compression* or *Model reduction* could be helpful.

4.1.3 HSS matrix transpose

The transpose of a HSS matrix will again be a HSS matrix. Suppose HSS matrix A is given by sequences B, R, W, U, V, D; it is quite easy to check that the HSS representation of its transpose A^H will be given by the sequences:

$$\begin{cases} \hat{D} = D^H & \hat{U} = V & \hat{V} = U \\ \hat{W}_l = R_l & \hat{W}_r = R_r & \hat{R}_l = W_l \\ \hat{R}_r = W_r & \hat{B}_u = B_l^H & \hat{B}_l = B_u^H \end{cases} \tag{4.14}$$

4.1.4 Generic inversion based on the state space representation

A state space representation for the inverse with the same state complexity can generically be given. We assume the existence of the inverse, the same hierarchical partitioning of the input and output vectors $\mathbf{x}$ and $\mathbf{b}$, and as generic conditions the invertibility of the direct operators $\mathbf{D}$ and $S = (I + \mathbf{B} Z_{\leftrightarrow} \mathbf{P}_{\text{leaf}}^H V^H D^{-1} U \mathbf{P}_{\text{leaf}})$, the latter being a (very) sparse perturbation of the unit operator with a local (that is leaf based) inversion operator. Let $\mathbf{L} = \mathbf{P}_{\text{leaf}}^H V^H D^{-1} U \mathbf{P}_{\text{leaf}}$, then we find

Theorem 1 *Under generic conditions, the inverse system T^{-1} has the following state space representation*

$$\begin{bmatrix} \mathbf{g} \\ \mathbf{f} \end{bmatrix} = \left\{ \begin{bmatrix} I & \\ & 0 \end{bmatrix} - \begin{bmatrix} \mathbf{L} \\ -I \end{bmatrix} S^{-1} \begin{bmatrix} B Z_{\leftrightarrow} & I \end{bmatrix} \right\} \left(\begin{bmatrix} Z_{\downarrow}^H W^H & \\ & R Z_{\downarrow} \end{bmatrix} \begin{bmatrix} \mathbf{g} \\ \mathbf{f} \end{bmatrix} + \begin{bmatrix} \mathbf{P}_{\text{leaf}}^H V^H D^{-1} \mathbf{b} \\ 0 \end{bmatrix} \right) \tag{4.15}$$

and output equation

$$\mathbf{x} = -\mathbf{D}^{-1} U \mathbf{P}_{\text{leaf}} \mathbf{f} + \mathbf{D}^{-1} \mathbf{b}. \tag{4.16}$$

Proof of the theorem follows from inversion of the output equation which involves the invertibility of the operator $\mathbf{D}$, and replacing the unknown $\mathbf{x}$ in the state equations, followed by a segregation of the terms that are directly dependent on the states and those that are dependent on the shifted states leading to the matrix $\begin{bmatrix} I & \mathbf{L} \\ -B Z_{\leftrightarrow} & I \end{bmatrix}$ whose inverse is easily computed as the first factor in the right hand side of the equation above. It should be remarked that this factor only involves operations at the leaf level of the hierarchy tree so that the given state space model can be efficiently executed (actually the inversion can be done using the original hierarchy tree much as is the case for the inversion of upper SSS systems) $\boxed{\cdot}$.

Having the theorem, we can derive a closed formula for T^{-1} assuming the generic invertibility conditions.

$$T^{-1} = \mathbf{D}^{-1} - \mathbf{D}^{-1}\mathbf{U}\mathbf{P}_{\text{leaf}}[I - \mathbf{R}Z_{\downarrow} + \mathbf{B}Z_{\leftrightarrow}(I - Z_{\downarrow}^{H}\mathbf{W}^{H})^{-1}\mathbf{P}_{\text{leaf}}^{H}\mathbf{D}^{-1}\mathbf{U}\mathbf{P}_{\text{leaf}}]^{-1}\mathbf{B}Z_{\leftrightarrow}(I - Z_{\downarrow}^{H}\mathbf{W}^{H})^{-1}\mathbf{P}_{\text{leaf}}^{H}\mathbf{V}^{H}\mathbf{D}^{-1} \tag{4.17}$$

The equation about is a compact diagonal representation of T^{-1}, it also proves that the inverse of an invertible HSS matrix is again a HSS matrix of comparable complexity.

4.1.5 LU decomposition of HSS matrix

Next, we establish that the LU factorization of a HSS matrix will consist of a product of two HSS matrices. One fast LU factorization method is described in [23], where the HSS representations of the two HSS matrices are given as:

Firstly, the intermediate variable[1] are defined recursively as follows: At a leaf,

$$g_{leaf} = V^{H}(D - UfV^{H})^{-1}U \tag{4.18}$$

at a node, let $\hat{B}_u = B_u - R_r f W_l^H$ and $\hat{B}_l = B_l - R_l f W_r^H$ in

$$g = \begin{bmatrix} W_l^H & W_r^H \end{bmatrix} \begin{bmatrix} g_l + g_l\hat{B}_l g_r \hat{B}_u g_l & -g_l\hat{B}_l g_r \\ -g_r\hat{B}_u g_l & g_r \end{bmatrix} \begin{bmatrix} R_l \\ R_r \end{bmatrix} \tag{4.19}$$

Note that, each of these f comes from the parent node. At the root node, we let $f = \phi$ and at a node:

$$f_l = R_l f W_l^H \tag{4.20}$$

When the computation on the left branch has returned, we compute:

$$f_r = \begin{bmatrix} R_l & B_u \end{bmatrix} \begin{bmatrix} f + fW_l^H g_l R_l f & -fW_l^H g_l \\ -g_l R_l f & g_l \end{bmatrix} \begin{bmatrix} W_r^H \\ B_l \end{bmatrix} \tag{4.21}$$

Once the computation of the intermediate variables has terminated, we have all the information needed to construct HSS representations for L and U factors.

let $L'U' = D - UfV^H$ be a LU decomposition at each leaf. Then the HSS representation of L factor will be given as: When at a node:

$$\begin{cases} \hat{R}_l = R_l & \hat{R}_r = R_r & \hat{W}_l = W_l \\ \hat{W}_r = W_r - w_l g_l \hat{B}_l & \hat{B}_u = \hat{B}_u & \hat{B}_l = 0 \end{cases} \tag{4.22}$$

When at a leaf:

$$\hat{U} = U \quad \hat{V} = (U')^{-H}V \quad D = L' \tag{4.23}$$

And the HSS representation of U factor will be given as: When at a node:

$$\begin{cases} \hat{R}_l = R_l & \hat{R}_r = R_r - \hat{B}_u g_l R_l & \hat{W}_l = W_l \\ \hat{W}_r = W_r & \hat{B}_u = 0 & \hat{B}_l = \hat{B}_l \end{cases} \tag{4.24}$$

When at a leaf:

$$\hat{U} = (L')^{-1}U \quad \hat{V} = V \quad D = U' \tag{4.25}$$

[1] the same notations for intermediate variables in section 4.1.2 are adapted.

4.2 Ancillary operations

In this section, we will discuss various ancillary operations that help to (re-) construct an HSS representation in various circumstances. These operations will help to reduce the HSS complexity or to keep the column base and row base dependencies of the HSS representation in algorithms that will be derived later. We also would like to point out that these ancillary operations described here are not complete; in [17], some other ancillary operations are described as well.

4.2.1 Column(Row) Bases insertion

When off-diagonal blocks have to be changed at the higher level nodes, column bases and row bases may have to be changed. To keep the column and row bases dependency, new column(Row) bases may have to be added to the lower level. We might be able to generate these bases from the column(Row) bases of the lower level nodes, but this is not guaranteed. Taking a conservative approach, We insert column(Row) bases into the lower level and the do a *compression* to reduce the complexity of HSS representation.

The algorithm combines two sub-algorithms (top-down base insertion and then a compression). The *compression* procedure is used to eliminate redundant bases and reduce the HSS complexity. *Compression* does not have to be done after every top-down Column(Row) Bases insertion. To save the computation cost, we may do one step of *compression* after a number of steps of bases insertion.

We will present row bases insertion in details, while column bases insertion can be easily derived from row bases insertion.

Top-down row bases insertion

Suppose that we need to add row bases v to one HSS node A without changing the matrix it represents (Note that, the row dimension of v should be the same as the column dimension of A.).The original HSS node is represented as

$$\begin{bmatrix} D_l & U_l B_u V_r^H \\ U_r B_l V_l^H & D_r \end{bmatrix}$$

This is the conventional HSS representation used in [6, 5, 4], the bases of its parent node can be generated from a combination of its bases and its translation matrices. Where its column bases and row bases will again be generated from its children. The algorithm should work in a top down fashion to modify the nodes and leaves in the HSS tree.

- **Row bases insertion at the node**

  We first split v according the column partition of A at this node: $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$. We then insert v_1 to its left child, v_2 to its right child using this algorithm recursively. Assuming the correctness of this algorithm, v_1 can be generated from D_l, and v_2 can be generated from D_r. We then need to modify the translation matrices of this node to make sure that the bases insertion does not change the matrix it represents. The translation matrices will be modified as follows: Where:

$$\begin{cases} \hat{W}_l = \begin{bmatrix} W_l & 0 \\ 0 & I \end{bmatrix} & \hat{W}_r = \begin{bmatrix} W_r & 0 \\ 0 & I \end{bmatrix} \\ \hat{B}_u = \begin{bmatrix} B_u & 0 \end{bmatrix} & \hat{B}_l = \begin{bmatrix} B_l & 0 \end{bmatrix} \end{cases} \tag{4.26}$$

- **Row bases insertion at the leaf**

  When we reach a leaf by recursion, we have to insert v to the leaf, we then let:

$$\hat{V} = \begin{bmatrix} V & v \end{bmatrix} \tag{4.27}$$

Compression

After applying top-down bases insertion to A, the row bases V required by the upper level can be generated from A. But the HSS representation we get is redundant, in the sense that the V matrices of the leaves may not be column orthonormal. It can be reduced by a modified version of Orthonormal Moves Algorithm presented in [17]. Since only V has been modified, we only have to factor V matrices as

$$V = \hat{V} w \tag{4.28}$$

This should be done by a rank revealing QR or QL factorization, then $\hat{V}$ will be column orthonormal (And it will surely be column independent). The factor w will then be propagated to the upper level, where the translation matrices B and W will be modified by the factor W as follows:

on the higher level, the translation matrices will be modified as:

$$\begin{cases} B_u' = B_u w_r^H & B_l' = B_l w_l^H \\ W_l' = w_l W_l & W_r' = w_r W_r \end{cases} \tag{4.29}$$

Where w_l comes from the left branch and w_r comes from the right branch.

Then we do compression on the higher level. Since only column bases have been modified, we only have to factor

$$\begin{bmatrix} W_l \\ W_r \end{bmatrix} = \begin{bmatrix} \hat{W}_l \\ \hat{W}_r \end{bmatrix} w \tag{4.30}$$

Note that: W_l and W_r have been modified by the w factors coming from its children.

This factorization can be the rank-revealing QR factorization. Again, the w factor will again be propagated to the upper level, and the algorithm proceeds recursively. For details, refer to [17] where the detailed algorithm with its data flow diagram is given.

After applying the whole algorithm to the a HSS node, the new row bases v will be inserted in this node. More precisely, it will be appended after its original row bases. Suppose the row bases of the original node is V, after inserting row bases v to this node, the row bases of the modified node will be $\begin{bmatrix} V & v \end{bmatrix}$. Note that: bases insertion does not change the HSS matrix, it only modifies its HSS representation.

Column bases insertion

The algorithm for column bases insertion is quite similar to the one for row bases insertion. Except the same modification will be done to U, R instead of V, W. And the B matrices will be modified as $B' = \begin{bmatrix} B \\ 0 \end{bmatrix}$ instead of $\begin{bmatrix} B & 0 \end{bmatrix}$ After applying the row bases insertion to a HSS node, the new column bases will be appended after its original column bases. The *compression* algorithm for column bases insertion should also be modified accordingly.

4.2.2 Append a matrix to a HSS matrix

This algorithm append a thin slice of matrix B to a HSS matrix A and we establish that the result of this operation will still be HSS matrix whose HSS representation can be computed easily. Obviously, we may append the matrix to the top of the HSS matrix, to the left of the HSS matrix, to the right of the HSS matrix or to the bottom of the HSS matrix. Here we just present the method to append matrix to the left of the HSS matrix. Others can be easily derived from this algorithm.

Append a rank-k matrix to a HSS matrix

Suppose

$$A' = \begin{bmatrix} B & A \end{bmatrix} \tag{4.31}$$

Matrix B should have the number of rows as A does, A is HSS matrix whose HSS representation is defined by sequences U_A, V_A, D_A, R_{Ar}, R_{Al}, W_{Ar}, W_{Al}, B_{Al} and B_{Ar}.

Instead of trying to absorb the B matrix into the HSS representation of A matrix, we rewrite the formula (4.31) as:

$$A' = \begin{bmatrix} - & - \\ B & A \end{bmatrix} \tag{4.32}$$

Where $-$ is the dummy matrix which has no rows. A' is an HSS matrix which has one more level than A does.

We then assume that $B = UB'V^H$. That is: B is a rank-k matrix. The decomposition of B can be computed by a URV factorization or SVD factorization. In practice, we normally have its decomposition.

Then the column bases U of B shall be inserted to the HSS representation of A so that U can be generated from the HSS representation of A. This can be done in many different ways. The most straightforward way is to insert the column bases using the algorithm in section 4.2.1 and then probably a step of *compression* depending on how many columns U has. Suppose that after column bases insertion, the HSS representation of A becomes $\hat{A}$. (Note that: column bases insertion does not change the HSS matrix, it only changes the HSS representation.)

Then A' will be represented as:

$$A' = \begin{bmatrix} - & - \\ UB'V^H & \hat{A} \end{bmatrix} \tag{4.33}$$

It is easy to check that the HSS representation of A' will be given as: when at the top node

$$\begin{cases} B_u = \emptyset & B_l = B' & W_l = | \\ W_r = \emptyset & R_l = \emptyset & R_r = | \end{cases} \tag{4.34}$$

when at the left branch:

$$D_l = - \quad U_l = \emptyset \quad V_l = V \tag{4.35}$$

when at the right branch:

$$D_r = \hat{A} \tag{4.36}$$

$|$ and $-$ represent dummy matrices with no columns or no rows. $\emptyset$ represents the dummy matrix without column or row. The dimensions of all these should be correct such that the HSS representation is still valid.

Matrix-append when bases are semi-separable

In practice, we almost never compute U and V, since these computations are costly and break the idea of HSS representation. For instance, when a matrix UBV^H needs to be appended to a HSS matrix A, U and V are not explicit stored. They are defined by the formulas (1.10) and (1.11).

In this case, the formulas in the last subsection will have to be modified accordingly, The left branch of A' will not be of just one level. Instead, the left child will be a sub-HSS tree defined by the following sequences: When at the root:

$$\begin{cases} B_u = \emptyset & B_l = B' & W_l = | \\ W_r = \emptyset & R_l = \emptyset & R_r = | \end{cases} \tag{4.37}$$

When at the nodes:

$$\begin{cases} \hat{R}_l = | & \hat{R}_r = | & \hat{B}_u = - \\ \hat{B}_l = - & \hat{W}_l = W_l & \hat{W}_r = W_r \end{cases} \tag{4.38}$$

When at the leaf:

$$\hat{U} = \emptyset \quad \hat{V} = V \quad \hat{D} = - \tag{4.39}$$

Note that: since the column bases U is also in a hierarchically semi-separable form, inserting it into A will be somewhat different than that in section (4.2.1). The modified formulas for inserting column bases U to A are given as:

$$\begin{cases} \hat{B}_u = \begin{bmatrix} B_{Au} \\ 0 \end{bmatrix} & \hat{B}_l = \begin{bmatrix} B_{Al} \\ 0 \end{bmatrix} \\ \hat{R}_l = \begin{bmatrix} R_{Al} & 0 \\ 0 & R_l \end{bmatrix} & \hat{R}_r = \begin{bmatrix} R_{Ar} & 0 \\ 0 & R_r \end{bmatrix} \quad \hat{U} = \begin{bmatrix} U_A & U \end{bmatrix} \end{cases} \tag{4.40}$$

4.3 Complexity analysis and numerical result

The main goal of this paper is to develop fast inverse algorithms for HSS matrices in which the elementary operations we defined will be extensively used. Before presenting the fast inverse algorithm and analyzing its computation complexity and characteristics of the inverse HSS matrix, we shall study the time complexity of these elementary operations. Their effect on HSS matrix will also be studied.

We shall also define the *representation complexity* of a HSS representation. That is, the same matrix can be represented by many different HSS representations, in which some are better than others in terms of computation complexity and space complexity. The *HSS representation complexity* should be defined in such a way that operation on the HSS representation with higher *HSS representation complexity* should cost more time and memory than that on HSS representations with lower *HSS representation complexity* does. Many indicators can be used. Here, we define the *HSS representation complexity* as follows

Definition 1 *HSS complexity: the total number of free entries in the HSS representation.*

Definition 2 *Free entries: free entries are the entries which can be changed without restriction (For instance, the number of free entries in $n \times n$ diagonal matrix will be n, that in $n \times n$ triangular matrix will be $n(n-1)/2$...etc).*

This figure actually indicates the least possible memory needed to store the HSS representation. It also implies the computation complexity, assuming each free entry is accessed once or for a small number of times during matrix operations on HSS representation.

4.3.1 Complexity analysis

Since most of these algorithms are not so complicated and some of these algorithms have been studied in other publications, we will list a summary table for existing HSS algorithms (including the ones described in this paper and the ones which are described in other publications)

We shall summarize these algorithms in terms of numerical complexity and their effect on HSS matrices(In particular, how the *HSS representation complexity* of the resulting HSS matrix will be affected). Then in the next section, we shall present the fast HSS inverse algorithms.

Operation with HSS	Numerical Complexity	HSS Complexity of the result
Matrix-Vector Multiplication[6]	$C_{Matrix\times Vector}(n) = O(n)$	The product b is a plain matrix
Matrix-Matrix Multiplication	$C_{Matrix\times Matrix}(n) = O(n)$	Increase additively
Construct HSS for rank-k matrix	$C_{k-construction}(n) = O(n)$	proportional to k
Bases insertion	$C_{Bases-insert}(n) = O(n)$	Increase by the size of V
Matrix-Append	$C_{Matrix-append}(n) = O(n)$	Increase by one level
Matrix addition	$C_{Addition}(n) = O(n)$	Increase additively
Compression	$C_{Compression}(n) = O(n)$	Does not increase
Model reduction[17]	$C_{Model-reduction}(n) = O(n)$	Decreases
LU Decomposition[23]	$C_{LU}(n) = O(n)$	Does not increase
Fast solve[6][4]	$C_{Solve}(n) = O(n)$	The result will be a plain matrix
Transpose	$C_{Transpose}(n) = O(n)$	Does not change

Table 4.1: Computation complexity analysis table

Decrease means the *HSS representation complexity* of the result will be decreased compared with the *HSS representation complexity* of the input HSS matrix.

Increase means the *HSS representation complexity* of the result will be Increased compared with the *HSS representation complexity* of the input HSS matrix.

Does not change means the *HSS representation complexity* of the result will not be higher than the *HSS representation complexity* of the input HSS matrix.

Increase by one level means the depth of the result HSS tree will be one level deeper than that of the input HSS matrix.

Increase by size of V means the *HSS representation complexity* of the result will be the *HSS representation complexity* of the input HSS matrix plus the size of V.

Increase additively means the *HSS representation complexity* of the result will be the sum of the *HSS representation complexity* of the two input HSS matrices.

4.3.2 Numerical result of the elementary operations

In this subsection, we will list the numerical results of some elementary operations excluding these which have been studied in [4, 5, 23, 17]. We shall study the computation time and memory usage versus the dimension of the HSS matrix. All HSS matrices are generated by a random HSS generator which takes four parameters (*dimension*, *level*, *granularity*, *balance*).

Dimension indicates the size of the HSS matrix, for simplicity, we only generate square matrix at this moment.

Level indicates the level of the HSS representation binary tree.

Granularity indicates the maximum size of translation matrices. Suppose *granularity* is 10, then all translation matrices can not be larger than 10×10 matrix. It also indicates the tree level implicitly, because the size of the leaves can not be larger than 10×10 matrices as well.

balance indicates whether the HSS matrix is uniformly partitioned. That is, whether the left children is of the same size as the right children. Again for simplicity, we only study *balanced* tree at this moment.

Numerical result of matrix operations concerning CPU time

We get the experiment result by doing HSS matrix operations on HSS matrices of different dimensions. The *granularity* of these HSS matrices will be given as 10 to guarantee the efficiency of these HSS matrix operations.

From the figure 4.1 we see that the computation time is somehow in linear with the dimension of the HSS matrix.(Or these curves all have linear upper-bound) The curves are somehow in a staircase shape because the HSS matrices of similar size generated by our HSS generator have the similar binary tree structure(for example, the same tree level) with different leaves. Computation on nodes counts for a large proportion of the whole computation while that on leaves plays a less important role. Since the adjacent HSS matrices have similar binary trees with translation matrices of comparable size, the difference on leaves does not affect computation time a lot. All time Vs dimension curves have a linear upper-bound.

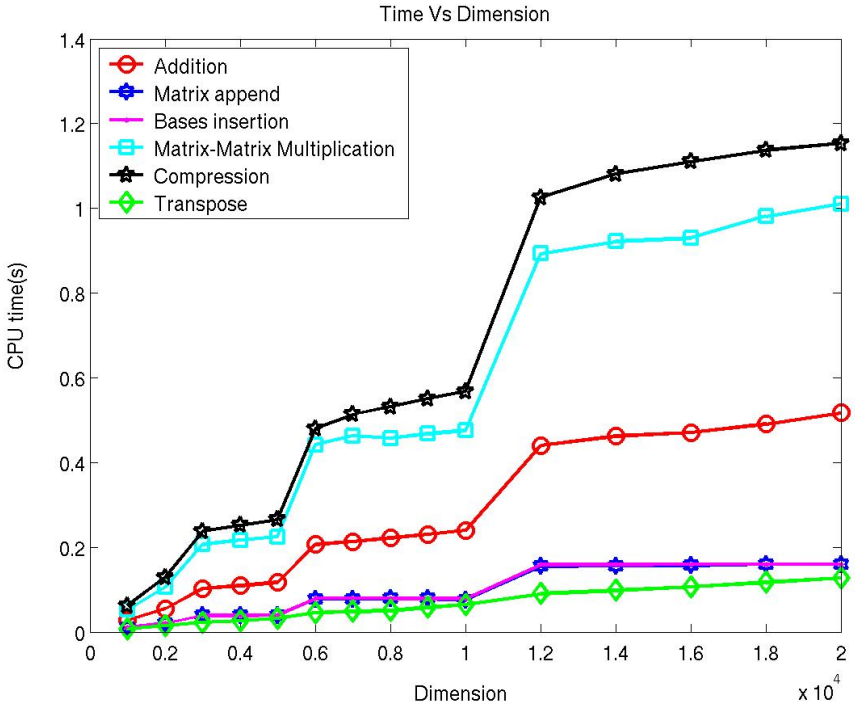

Figure 4.1: Time Vs Dimension

Numerical result of matrix operations concerning memory usage

With the ever increasing size of problems, memory usage becomes more and more important. Under some circumstance when the memory requirement of the algorithm can hardly be satisfied, it is even

more important than CPU time. We can of course attack this problem by investing more hardware. GRID computing these days has become an appealing approach. However, even on a GRID with thousands of nodes, a memory inefficient algorithm can easily run out of steam. We take the same set of numerical experiments as these in the last subsection and study how much memory these algorithms will take. It is

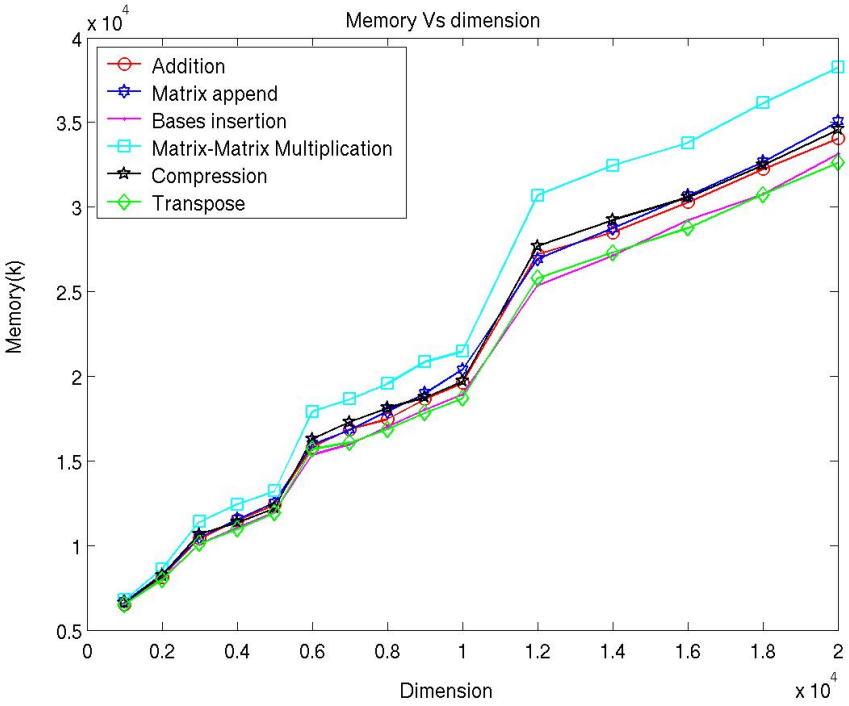

Figure 4.2: Memory Vs Dimension

quite clear from figure 4.2 that the memory complexity of these matrix operation with HSS representation is in linear with the dimension of the HSS matrix(assuming small translation matrices).

4.4 The HSS Matrix inverse

After presenting all these elementary algorithms, we are ready to construct the inverse of a HSS matrix. We establish that the inverse of a HSS matrix will be another HSS matrix. We shall also study the HSS complexity of the inverse HSS matrix compared with the original one.

4.4.1 Fast HSS matrix inverse algorithm based on Schur complement

In this section we present a fast inverse algorithm based on Schur complement, we shall also point out that, since this algorithm is based on computing Schur complement recursively, it only works when the Schur complements exist and invertible. If not, the inverse algorithm will break. However, in [7], a URV factorization method for HSS matrix is presented. The URV factorization algorithm will help us generalize this algorithm to any invertible matrix and even to non-invertible matrix whose Moore-Penrose inverse can be computed.

Let:

$$A = \begin{bmatrix} D_l & U_l B_u V_r^H \\ U_r B_l V_l^H & D_r \end{bmatrix}$$

Where D_l and D_r could also be HSS matrices. For the ease of analyzing computation complexity, we assume A to be a HSS matrix with p level of depth. D_l and D_r are of $p-1$ level. And the HSS matrix A is uniformly partitioned, which means the dimension of D_l and D_r are half of the dimension of A. All on-diagonal blocks are square and invertible. For efficiency of this algorithm, we also assume that all translation matrices to be small or thin matrices, which is normally the case for all HSS algorithms. However, these assumptions are made for the sake of the efficiency of the algorithm and ease of complexity analysis, not for the correctness of this algorithm.

Applying Inverse operation with Schur complement S, the inverse of A is:

$$A^{-1} = \begin{bmatrix} D_l^{-1} + D_l^{-1} U_l B_u V_r^H S^{-1} U_r B_l V_l^H D_l^{-1} & -D_l^{-1} U_l B_u V_r^H S^{-1} \\ -S^{-1} U_r B_l V_l^H D_l^{-1} & S^{-1} \end{bmatrix} \tag{4.41}$$

Where $S = D_r - U_r B_l V_l^H D_l^{-1} U_l B_u V_r^H$. We first establish that S will be a HSS matrix. We can rewrite the above formula as:

$$S = D_r - S'$$

where

$$S' = U_r B_l V_l^H D_l^{-1} U_l B_u V_r^H$$

let's assume the correctness of this inverse algorithm, the D_l^{-1} has already been defined.

Because D_l^{-1} is hierarchically semi-separable, $U_r B_l V_l^H$ and $U_l B_u V_r^H$ are rank-k matrices with hierarchically semi-separable bases, the product $S' = U_r B_l V_l^H D_l^{-1} U_l B_u V_r^H$ can be computed using the HSS matrix-matrix multiplication with rank-k matrix presented before. Taking the straightforward implementation, this takes one HSS inverse, two rank-k matrix HSS construction, and two HSS Matrix-Matrix multiplications. It amounts to $C_{inverse}(n/2) + 2C_{k-construction}(n/2) + 2C_{Matrix \times Matrix}(n/2)$ operations.

The product S' will be a hierarchically semi-separable matrix. We then compute $S = D_r - S'$, which takes one HSS addition operation. Because the row bases U_r and column bases V_r can be generated recursively from D_r, the HSS complexity of S can be brought down to that of D_r by the compression operation. These two operations amount to $C_{addition}(n/2) + C_{Compression}(n/2)$ computation.

Then we should invert S to get S^{-1} using the inverse algorithm which takes $C_{inverse}(n/2)$.

Next we compute $-S^{-1} U_r B_l V_l^H D_l^{-1}$ and $-D_l^{-1} U_l B_u V_r^H S^{-1}$. For $-S^{-1} U_r B_l V_l^H D_l^{-1}$ let's rewrite the formula as

$$\begin{bmatrix} -S^{-1} & | \\ - & \emptyset \end{bmatrix} \begin{bmatrix} | & U_r B_l V_l^H \\ \emptyset & - \end{bmatrix} \begin{bmatrix} \emptyset & - \\ | & D_l^{-1} \end{bmatrix} = \begin{bmatrix} | & -S^{-1} U_r B_l V_l^H D_l^{-1} \\ \emptyset & - \end{bmatrix} \tag{4.42}$$

This operation amounts to two matrix-matrix multiplication. What's good about this form is that: we get again a semi-separable column bases , a matrix B and a semi-separable row bases ($\hat{U}_l \hat{B}_u \hat{V}_r^H$) in the new HSS representation. Operation for $-D_l^{-1} U_l B_u V_r^H S^{-1}$ is similar. This step amounts to $4C_{Matrix \times Matrix}(n/2)$ operations.

It remains to compute the HSS representation of the left upper block in A^{-1}, since D_l^{-1}, $U_l B_u V_r^H$, S^{-1} and $U_r B_l V_l^H$ are all hierarchically semi-separable, efficient HSS Matrix-Matrix multiplication algorithm can be used Also, because some of the terms in this product have been computed, the result can be reused. to compute the product $C = D_l^{-1} U_l B_u V_r^H S^{-1} U_r B_l V_l^H D_l^{-1}$, and the product will again be

HSS matrix. This computation step will take optimally one HSS Matrix-Matrix multiplication, which amounts to $C_{Matrix\times Matrix}(n/2)$.

then we add C and D_l^{-1} using HSS matrix addition algorithm, and the sum $\hat{D}_l = D_l^{-1} + C$ will be another HSS matrix. Again, we can easily see that the bases of C can be generated from bases of D_l^{-1}, thus, a compression operation can bring the HSS complexity back to that of D_l^{-1}. This step takes $C_{addition}(n/2) + C_{Compression}(n/2)$

At this point we already know the left upper block $\hat{D}_l$, the right bottom block S^{-1} and the off-diagonal blocks $\hat{U}_r\hat{B}_l\hat{V}_l^H$ and $\hat{U}_l\hat{B}_u\hat{V}_r^H$. Then we need to do column and row bases insertion to keep the row and column base dependency, taking the conservative approach, we can always insert all the bases into the left upper and right bottom HSS blocks, even though some of these bases could be generated from the bases of the on-diagonal blocks. We insert row bases $\hat{U}_l$ and column bases $\hat{V}_l$ to $\hat{D}_l$ (This algorithm is described in section 2). This modifies $\hat{D}_l$ to be D_l. Row bases $\hat{U}_r$ and column bases $\hat{V}_r$ are inserted to S^{-1}, let the HSS representation of S^{-1} after bases insertion operations be $\hat{D}_r$. This step takes $4C_{bases-insert}(n/2)$.

At last a compression or model reduction may be used to bring down the HSS complexity. And we can prove that the complexity will be brought back to the sum of the HSS complexity of D_l^{-1}, S^{-1} and the root node. Since the complexity of the top node stays the same (because B_u and B_l remains the same), the HSS complexity of the inverse HSS matrix will be proportional to that of the original HSS matrix. The compression operation will take $C_{Compression}(n)$ computation.

After assemble all the computations, the computation cost will be defined recursively as:

$$C_{inverse}(n) = 2C_{inverse}(n/2) + 7C_{Matrix\times Matrix}(n/2) + 2C_{addition}(n/2)+$$

$$2C_{k-construction}(n/2) + 2C_{Compression}(n/2) + C_{Compression}(n)+$$

$$4C_{bases-insert}(n/2)$$

Since we know the computation complexity of the elementary operations, the formula above can be written as:

$$C_{inverse}(n) = 2C_{inverse}(n/2) + 19O(n/2) = O(n\log(n)) \tag{4.43}$$

After all these recursive computation, we have all the ingredient needed to construct the HSS representation of A^{-1}:

We define the HSS tree recursively as:

$$\begin{cases} \tilde{D}_l = \hat{D}_l & \tilde{D}_r = \hat{D}_r & \hat{B}_u = B_u \\ \hat{B}_l = B_l & \hat{R}_l = I & \hat{R}_r = I \\ \hat{W}_l = I & \hat{W}_r = I \end{cases} \tag{4.44}$$

4.4.2 Fast inverse algorithm based on LU factorization

The fast inverse algorithm in the above section may be awkward to implement because of all these ad-hoc operations to reduce the complexity of HSS representation. In order to simplify the fast Schur inverse algorithm we can apply a LU factorization[2] first(The algorithm for HSS LU decomposition has been described in section 4.1.5), then these two factors are block upper-triangular and block lower-triangular matrices which are much easier to invert. (Note that: since we just need these two factors to be block upper-triangular matrix and block lower-triangular matrix, the LU decomposition described can be simplified)

Now suppose we have a HSS matrix A which is defined by sequences B, R, W, U, V, D. The problem is how to compute its inverse A^{-1}.

The first step is to do LU factorization for A matrix. Then we have two block triangular factors namely A_l,A_u, where A_l is block lower-triangular, A_u is block upper-triangular. We can invert these two matrices separately and finally multiply them together if we want. But it is not really necessary to multiply them together since it is advantageous to have two "smaller" HSS matrices (because the computation complexity may go up cubically with the size of the translation matrices).

Here we only study the inverse of A_u while the inverse of A_l is quite similar. Let A_u is defined by sequences $\hat{B}, \hat{R}, \hat{W}, \hat{U}, \hat{V}, \hat{D}$. Then for the first level of its HSS representation we can write it as:

$$A_u = \begin{bmatrix} \hat{D}_l & \hat{U}_l \hat{B}_u \hat{V}_r^H \\ \hat{U}_r \hat{B}_l \hat{V}_l^H & \hat{D}_r \end{bmatrix}$$

However, since it is upper triangular, the lower-left block should be all zeros. and D_l, D_r are upper-triangular matrices again.

then let's have a look at the simplified inverse formula based on Schur complement. the inverse of A is:

$$A_u^{-1} = \begin{bmatrix} \hat{D}_l^{-1} & -\hat{D}_l^{-1} \hat{U}_l \hat{B}_u \hat{V}_r^H \hat{D}_r^{-1} \\ 0 & \hat{D}_r^{-1} \end{bmatrix} \tag{4.45}$$

We need to compute D_l^{-1} and D_r^{-1} which are all upper-triangular matrices. Assuming the correctness of this inverse algorithm, it can be applied recursively on $\hat{D}_l$ and $\hat{D}_r$, then we only need to compute the upper-right block $-\hat{D}_l^{-1} \hat{U}_l \hat{B}_u \hat{V}_r^H \hat{D}_r^{-1}$ and to keep the HSS representation valid, we need to have the product in UBV^H form, where U and V are HSS compatible (this can be done using the trick described in the last subsection). Then we shall insert the row bases(V) and column bases(U) into the lower HSS blocks. A step of compression of model reduction is useful to bring down the HSS complexity.

About the computation complexity, since the computation complexity for fast LU factorization method is $O(n)$ and it is easy to see the complexity of the inverse algorithm is $O(n\log n)$, the computation complexity of this algorithm is still $O(n\log n)$. The advantage of this algorithm is that it needs much less HSS matrix operations each iteration, thus the HSS complexity of the inverse does not increase a lot; this also leads to less *compression* steps. Another benefit is that here we get two smaller HSS matrices instead of a big one. This reduces the computation complexity of further operations.

4.5 Connection among SSS, HSS and time varying notation

In the earlier papers on SSS [35, 33], lots of efficient algorithms have been developed. Then in order to apply multi-way solver, a more efficient but less flexible representation HSS is presented in[5, 6, 4]. Although different algorithms have to be used corresponding to these two seemingly different representations, we would like to show that they are not so different, and we will show how these two representations can be converted to each other. By converting between these two representations, we can take advantages of the fast algorithms for these two different representations.

4.5.1 From SSS to HSS

In [33], the SSS representation for A is defined as follows:

Let A be an $N \times N$ matrix satisfying the SSS matrix structure. Then there exist n positive integers $m_1, ...m_n$ with $N = m_1 + ...m_n$ to block-partition A as $A = A_{i,j}$, where $A_{ij} \in C^{m_i \times m_j}$ satisfies

$$A_{ij} = \begin{cases} D_i & \text{if } i = j \\ U_i W_{i+1} ... W_{j-1} V_j^H & \text{if } j > i \\ P_i R_{i-1} ... R_{j+1} Q_j^H & \text{if } j < i \end{cases} \tag{4.46}$$

For simplicity, we consider casual operators. For $n = 4$, the matrix A has the form

$$A = \begin{bmatrix} D_1 & U_1 V_2^H & U_1 W_2 V_3^H & U_1 W_2 W_3 V_4^H \\ 0 & D_2 & U_2 V_3^H & U_2 W_3 V_4^H \\ 0 & 0 & D_3 & U_3 V_4^H \\ 0 & 0 & 0 & D_4 \end{bmatrix} \tag{4.47}$$

Let's first split the SSS matrix as following

$$A = \left[\begin{array}{cc|cc} D_1 & U_1 V_2^H & U_1 W_2 V_3^H & U_1 W_2 W_3 V_4^H \\ 0 & D_2 & U_2 V_3^H & U_2 W_3 V_4^H \\ \hline 0 & 0 & D_3 & U_3 V_4^H \\ 0 & 0 & 0 & D_4 \end{array}\right] \tag{4.48}$$

The top-left block goes to the left branch of the HSS representation, while the right-bottom block goes to the right branch. Then, we can construct the root by setting:
For the top HSS node:

$$\begin{cases} B_u = I & B_l = 0 & W_l = I \\ W_r = W_2^H & R_l = W_3 & R_r = I \end{cases} \tag{4.49}$$

Then we construct the left branch with the similar partitioning.

$$\left[\begin{array}{c|c} D_1 & U_1 V_2^H \\ \hline 0 & D_2 \end{array}\right] \tag{4.50}$$

It is then quite clear that: for the one level lower left child,

$$D_{ll} = D_1 \quad U_{ll} = U_1 \quad V_{ll} = V_1 \tag{4.51}$$

while for the right child

$$D_{lr} = D_2 \quad U_{lr} = U_2 \quad V_{lr} = V_2 \tag{4.52}$$

In order to keep the HSS representation valid, R and W matrices on the left node should be set properly. That is: for the left node, let:

$$\begin{cases} R_{ll} = W_2 & R_{lr} = I & W_{ll} = I \\ W_{lr} = W_1^H & B_{ll} = 0 & B_{lu} = I \end{cases} \tag{4.53}$$

Then we consider the right branch with the similar partitioning as in (4.50). Quite similarly, we can get the right branch as: For the one level lower left child,

$$D_{rl} = D_3 \quad U_{rl} = U_3 \quad V_{rl} = V_3 \tag{4.54}$$

while for the right child

$$D_{rr} = D_4 \quad U_{rr} = U_4 \quad V_{rr} = V_4 \tag{4.55}$$

In order to keep the HSS representation valid, R and W matrices on the right node should be set properly. That is: for the right node, let:

$$\begin{cases} R_{rl} = W_4 & R_{rr} = I & W_{rl} = I \\ W_{rr} = W_3^H & B_{rl} = 0 & B_{ru} = I \end{cases} \tag{4.56}$$

Finally the HSS representation of the same matrix can be written as:

$$A = \begin{bmatrix} D_{ll} & U_{ll}B_{lu}V_{lr}^H & U_{ll}R_{ll}B_uW_{rl}^HV_{rl}^H & U_{ll}R_{ll}B_uW_{rr}^HV_{rr}^H \\ 0 & D_{lr} & U_{lr}R_{lr}B_uW_{rl}^HV_{rl}^H & U_{lr}R_{lr}B_uW_{rr}^HV_{rr}^H \\ 0 & 0 & D_{rl} & U_{rl}B_{ru}V_{rr}^H \\ 0 & 0 & 0 & D_{rr} \end{bmatrix} \tag{4.57}$$

with all the translation matrices set in equation (4.49) to (4.56).

After illustrating the algorithm with the above 2-level HSS example, we are ready to introduce the former algorithm:

Given the SSS matrix represented in the form (4.46). For simplicity again, we only consider casual SSS operator that is block upper-triangular.

First we need to partition the SSS matrix according to certain hierarchical partitioning. Then for the current HSS node which should contain the SSS blocks A_{xy} where $i \le x, y \le j$ ($1 \le i < j \le n$). Assuming the HSS block is further partitioned at k ($i < k < j$). the translation matrices of the current node can be chosen as:

$$\begin{cases} B_u = I & B_l = 0 & W_l = I \\ W_r = \prod_{x=k}^{i} W_x^H & R_l = \prod_{x=k+1}^{j} W_x & R_r = I \end{cases} \tag{4.58}$$

note that: for undefined W_x matrices, we let it be I (the dimension of I is defined according to context). While, if $i = k$ or $k + 1 = j$, then one(or two) HSS leaf (leaves) has to be constructed by letting

$$D_{\text{HSS}} = D_k \quad U_{\text{HSS}} = U_k \quad V_{\text{HSS}} = V_k \tag{4.59}$$

After constructing the HSS node of the current level, the same algorithm is applied recursively to construct HSS node for SSS blocks $A_{xy}, i \le x, y \le k$ and that for SSS block $A_{xy}, k + 1 \le x, y \le j$ (the recursion stops when a leaf is constructed.).

Observing the facts that all B_l matrices are zeros matrices and W_l,R_r are identity matrices, modification can be done to get a more efficient HSS representation.

After presenting this conversion algorithm[3], it is safe to conclude that HSS is a special case of SSS

[3]Actually a more generic algorithm can be drawn by equalizing the compact diagonal representations for SSS and HSS of the same matrix A:

$$\mathbf{D}_{\text{SSS}}+\mathbf{U}_{\text{SSS}}Z(I-\mathbf{W}_{\text{SSS}}Z)^{-1}\mathbf{V}_{\text{SSS}} = \mathbf{D}_{\text{HSS}}+\mathbf{U}_{\text{HSS}}\mathbf{P}_{\text{leaf}}(I-\mathbf{R}_{\text{HSS}}Z_{\downarrow})^{-1}\mathbf{B}_{\text{HSS}}Z_{\leftrightarrow}(I-Z_{\downarrow}^H\mathbf{W}_{\text{HSS}}^H)^{-1}\mathbf{P}_{\text{leaf}}\mathbf{V}_{\text{HSS}}^H$$

It is evident that $\mathbf{D}_{\text{HSS}} = \mathbf{D}_{\text{SSS}}$, $\mathbf{U}_{\text{HSS}} = \mathbf{U}_{\text{SSS}}$ and $\mathbf{V}_{\text{HSS}} = \mathbf{V}_{\text{SSS}}$. Then we just have to study the relation between $\mathbf{W}_{\text{SSS}}$ and $\mathbf{R}_{\text{HSS}}$, $\mathbf{W}_{\text{HSS}}$, $\mathbf{B}_{\text{HSS}}$ by solving the equation:

$$Z(I - \mathbf{W}_{\text{SSS}}Z)^{-1} = \mathbf{P}_{\text{leaf}}(I - \mathbf{R}_{\text{HSS}}Z_{\downarrow})^{-1}\mathbf{B}_{\text{HSS}}Z_{\leftrightarrow}(I - Z_{\downarrow}^H\mathbf{W}_{\text{HSS}}^H)^{-1}\mathbf{P}_{\text{leaf}}$$

Complicated as this equation is, simplification needs to be made. One possible simplification which is adapted in the conversion algorithm presented in section 4.5.1 is to hold $B_{HSSu} = I$, $B_{HSSl} = 0$ and hold some W_{HSS} and R_{HSS} to be I. Then this equation becomes easier to solve. Other approaches are possible. The principle of these simplifications is to solve the equation without matrix inverse operation.

with more restrictions . In fact, this is also one of the conclusions made in [17].

4.5.2 From HSS to Time-varying notation(or equivalently SSS)

In this section we will present an algorithm to convert a HSS representation to time-varying notation. The conversion algorithm exploits maximal parallelism and represents these parallel computation stepwise. Thus a sequential computational network can be found for HSS matrix-vector multiplication. This sequential computational network can then be represented using Time-varying notation (or equivalently SSS).

However, in most Time-varying literatures, people work on left operators; while under some circumstances, we feel more comfortable with right operators. All we need to do is to match the low triangular matrix(T as in $b = Tx$) with the corresponding causal operator($\hat{T}$ as in $\hat{b} = \hat{x}\hat{T}$) such that $T = \hat{T}^H$, $x = \hat{x}^H$, $b = \hat{b}^H$.

We consider the formula (1.13) and as iterative formula to update the state variable g, the iterative equation as well as the output equation will be written as [4]:

$$\begin{cases} \mathbf{g}_i = \mathbf{P}_{\text{leaf}}^H \mathbf{V}_{\text{HSS}}^H \mathbf{x} + Z_{\downarrow}^H \mathbf{W}_{\text{HSS}}^H \mathbf{g}_{i-1} \\ b_i = \phi \end{cases} \tag{4.60}$$

We can unroll equation (4.60) so that it becomes:

$$\begin{cases} \mathbf{g}_i = \sum_{k=1}^{i} (Z_{\downarrow}^H \mathbf{W}_{\text{HSS}}^H)^{k-1} \mathbf{P}_{\text{leaf}}^H \mathbf{V}_{\text{HSS}}^H \mathbf{x} \\ b_i = \phi \end{cases} \tag{4.61}$$

Easy to see that $\mathbf{g}_i$ is the sum of $\mathbf{g}_{i-1}$ and vector $(Z_{\downarrow}^H \mathbf{W}_{\text{HSS}}^H)^{i-1} \mathbf{P}_{\text{leaf}}^H \mathbf{V}_{\text{HSS}}^H \mathbf{x}$:

$$\mathbf{g}_i = \mathbf{g}_{i-1} + (Z_{\downarrow}^H \mathbf{W}_{\text{HSS}}^H)^{i-1} \mathbf{P}_{\text{leaf}}^H \mathbf{V}_{\text{HSS}}^H \mathbf{x} \tag{4.62}$$

Because of these shift operators, terms in (4.61) do not overwrite each other in $\mathbf{g}$.

Since $(Z_{\downarrow}^H \mathbf{W}_{\text{HSS}}^H)^{i-2} \mathbf{P}_{\text{leaf}}^H \mathbf{V}_{\text{HSS}}^H \mathbf{x}$ is contained in $\mathbf{g}_{i-1}$, the last term in (4.62) can be generated from $\mathbf{g}_{i-1}$. Now, we need to modify $\mathbf{W}_{\text{HSS}}$ as $\mathbf{W}_{\text{HSS}_i}$:

$$\mathbf{W}_{\text{HSS}i(jk)} = \begin{cases} \mathbf{W}_{\text{HSS}(jk)} & \text{when entry}(jk) \text{ is the entry of } W_{i;l} \text{ in } \mathbf{W}_{\text{HSS}} \\ 0 & \text{else} \end{cases}$$

Then equation (4.62) can be represented as:

$$\mathbf{g}_i = \begin{cases} (I + Z_{\downarrow}^H \mathbf{W}_{\text{HSS}_i}^H) \mathbf{g}_{i-1} & i \geq 2 \\ \mathbf{P}_{\text{leaf}}^H \mathbf{V}_{\text{HSS}}^H \mathbf{x} & i = 1 \end{cases} \tag{4.63}$$

Now let us review the time-varying state equations in [9]:

$$\begin{cases} x_{k+1} = x_k A_k + u_k B_k \\ y_k = x_k C_k + u_k D_k \end{cases}$$

[4]The compact block diagonal representation is used in this subsection. Leaf projection operator P_{leaf} and shift operator $Z_{\downarrow}$ $Z_{\leftrightarrow}$ are also extensively used. A brief description can be found in section 1.2, however, it maybe insufficient; for details, refer to [4]

the formula above might be confusing, since the notations in [9] are defined quite differently. Note that in the above equation, x represents states, y represents outputs, and u represents inputs. While x is defined as inputs, y is defined as outputs for the rest of this paper.

Then it is easy to set the computational model of the first k iterations as (k is the depth of the corresponding HSS tree):

$$\left[\begin{array}{c|c} A_i & C_i \\ \hline B_i & D_i \end{array}\right] = \begin{cases} \left[\begin{array}{cc} - & \phi \\ \mathbf{V}_{\text{HSS}}\mathbf{P}_{\text{leaf}} & | \end{array}\right] & i = 1 \\ \left[\begin{array}{cc} (I + \mathbf{W}_{\text{HSS}i} Z_{\downarrow}) & | \\ - & \phi \end{array}\right] & 2 \le i \le k \end{cases} \tag{4.64}$$

similarly, equation (1.13) as well as the output equation can be written in an iterative fashion:

$$\begin{cases} \mathbf{f}_i = \mathbf{R}_{\text{HSS}} Z_{\downarrow} \mathbf{f}_{i-1} + \mathbf{B}_{\text{HSS}} Z_{\leftrightarrow} \mathbf{g}_k \\ b_i = \phi \end{cases} \tag{4.65}$$

similarly, we can unroll the above iterative equation so that it becomes:

$$\begin{cases} \mathbf{f}_i = \sum_{k=1}^{i} (\mathbf{R}_{\text{HSS}} Z_{\downarrow})^{k-1} \mathbf{B}_{\text{HSS}} Z_{\leftrightarrow} \mathbf{g}_k \\ b_i = \phi \end{cases} \tag{4.66}$$

This equation is almost the same as equation (4.61); with the same approach, we defined $\mathbf{R}_{\text{HSS}_i}$ as:

$$\mathbf{R}_{\text{HSS}i(jk)} = \begin{cases} \mathbf{R}_{\text{HSS}(jk)} & \text{when entry}(jk) \text{ is the entry of } R_{k-i+1;l} \text{ in } \mathbf{R}_{\text{HSS}} \\ 0 & \text{else} \end{cases}$$

Then the computational model from step $k+1$ to $2k$ can be represented as:

$$\left[\begin{array}{c|c} A_i & C_i \\ \hline B_i & D_i \end{array}\right] = \begin{cases} \left[\begin{array}{cc} Z_{\leftrightarrow}^{H} \mathbf{B}_{\text{HSS}}^{H} & | \\ - & \phi \end{array}\right] & i = k+1 \\ \left[\begin{array}{cc} (I + Z_{\downarrow}^{H} \mathbf{R}_{\text{HSSi}}^{\mathbf{H}}) & | \\ - & \phi \end{array}\right] & k+2 \le i \le 2k \end{cases} \tag{4.67}$$

after $2k$ steps $\mathbf{f}_k$ is available, the output b can be computed using the formula:

$$\mathbf{b} = \mathbf{D}_{\text{HSS}}\mathbf{x} + \mathbf{U}_{\text{HSS}}\mathbf{P}_{\text{leaf}}\mathbf{f}_k$$

Then the computational model for the last step will be

$$\left[\begin{array}{c|c} A_i & C_i \\ \hline B_i & D_i \end{array}\right] = \left[\begin{array}{cc} | & \mathbf{P}_{\text{leaf}}{}^{H} \mathbf{U}_{\text{HSS}}{}^{H} \\ | & \mathbf{D}_{\text{HSS}}{}^{H} \end{array}\right] \text{ where } i = 2k+1 \tag{4.68}$$

Here we only use the block diagonal form to represent each computational step in a concise way(the dimensions of these big block diagonal matrices are either defined by the construction rules or by the context).

To summarize this conversion algorithm, we put assemble (4.64), (4.67), (4.68) together as:

$$\left[\begin{array}{c|c} A_i & C_i \\ \hline B_i & D_i \end{array}\right] = \begin{cases} \left[\begin{array}{cc} - & \phi \\ \mathbf{V}_{\text{HSS}}\mathbf{P}_{\text{leaf}} & | \end{array}\right] & i = 1 \\ \left[\begin{array}{cc} (I + \mathbf{W}_{\text{HSS}i} Z_{\downarrow}) & | \\ - & \phi \end{array}\right] & 2 \le i \le k \\ \left[\begin{array}{cc} Z_{\leftrightarrow}^{H}\mathbf{B}_{\text{HSS}}^{H} & | \\ - & \phi \end{array}\right] & i = k+1 \\ \left[\begin{array}{cc} (I + Z_{\downarrow}^{H}\mathbf{R}_{\text{HSSi}}^{\mathbf{H}}) & | \\ - & \phi \end{array}\right] & k+2 \le i \le 2k \\ \left[\begin{array}{cc} | & \mathbf{P}_{\text{leaf}}{}^{H}\mathbf{U}_{\text{HSS}}{}^{H} \\ | & \mathbf{D}_{\text{HSS}}{}^{H} \end{array}\right] & i = 2k+1 \end{cases} \tag{4.69}$$

When comes to the realization of the computational model, the following rules and facts should be used:

- Since time-varying computational network actually computes $\hat{b} = \hat{x}T^H$. Input vector x should be transposed before fed into the network, and the output vector is the transpose of the b. It is also quite evident that only the first stage and the last stage need input x (the same x). So input vector x should to be duplicated.

- Iterative equation (4.60), (4.65) and computational model (4.69) specify the **relation** between successive states. In other words, they specify **what** is done in these computational steps instead of **how** it is done. An efficient realization of this computational model is described with the Matrix-Vector multiplication algorithm in [5, 23, 17]

- During each computational step, not every entries of g and f will be used. These entries that are not needed in the succeeding steps can be discarded or overwritten.

4.5.3 Recursive time-varying notation for HSS

In this section, we shall consider HSS as recursive SSS. Since SSS can be represented by the concise time-varying notation in [9] . We will present how HSS representation can be represented by recursive time-varying notation. We shall first illustrate the algorithm by an example on 8×8 HSS representation.

With the re-partitioning illustrated from figure 4.3 to figure 4.4.
the 4-level balanced HSS representation is rewritten as:

$$A = \begin{bmatrix} D_l & U_l B_u W_{rl}^H V_{rl}^H & U_l B_u W_{rr}^H W_{rrl}^H V_{rrl}^H & U_l B_u W_{rr}^H W_{rrr}^H V_{rrr}^H \\ U_{rl} R_{rl} B_l V_l^H & D_{rl} & U_{rl} B_{ru} W_{rrl}^H V_{rrl}^H & U_{rl} B_{ru} W_{rrr}^H V_{rrr}^H \\ U_{rrl} R_{rrl} R_{rr} B_l V_l^H & U_{rrl} R_{rrl} B_{rl} V_{rl}^H & D_{rrl} & U_{rrl} B_{rru} V_{rrr}^H \\ U_{rrr} R_{rrr} R_{rr} B_l V_l^H & U_{rrr} R_{rrr} B_{rl} V_{rl}^H & U_{rrr} B_{rrl} V_{rrl}^H & D_{rrr} \end{bmatrix} \tag{4.70}$$

Since time varying system works on upper triangular matrices, we assume all lower triangular part be 0.

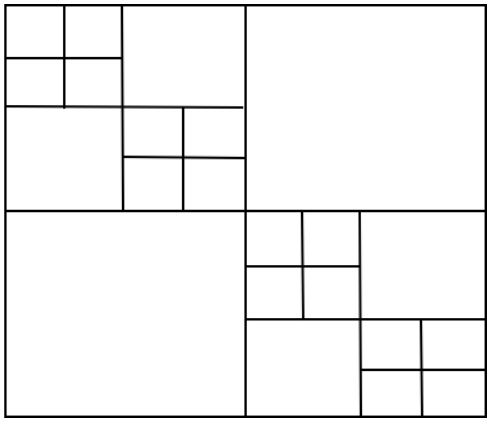

Figure 4.3: HSS partitioning

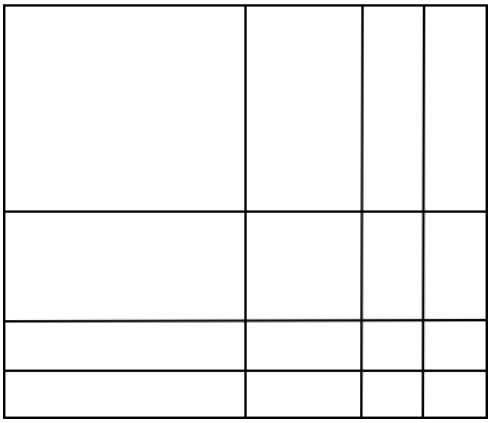

Figure 4.4: SSS partitioning

then we get HSS representation as:

$$A = \begin{bmatrix} D_l & U_l B_u W_{rl}^H V_{rl}^H & U_l B_u W_{rr}^H W_{rrl}^H V_{rrl}^H & U_l B_u W_{rr}^H W_{rrr}^H V_{rrr}^H \\ 0 & D_{rl} & U_{rl} B_{ru} W_{rrl}^H V_{rrl}^H & U_{rl} B_{ru} W_{rrr}^H V_{rrr}^H \\ 0 & 0 & D_{rrl} & U_{rrl} B_{rru} V_{rrr}^H \\ 0 & 0 & 0 & D_{rrr} \end{bmatrix} \tag{4.71}$$

Now we have a look at a time varying notation when $k = 4$:

$$A = \begin{bmatrix} D_1 & B_1 C_2 & B_1 A_2 C_3 & B_1 A_2 A_3 C_4 \\ 0 & D_2 & B_2 C_3 & B_2 A_3 C_4 \\ 0 & 0 & D_3 & B_3 C_4 \\ 0 & 0 & 0 & D_4 \end{bmatrix} \tag{4.72}$$

Then we immediately see similarities between the above two representations. Knowing the HSS representation, it is not difficult to set up all the unknown matrices in the latter time varying notation. After setting up all the unknown matrices, the 4 step realization ($\left[\begin{array}{c|c} A & C \\ \hline B & D \end{array}\right]$) of this time varying system

can be collected as:

$$T_1 = \left[\begin{array}{c|c} \cdot & \cdot \\ \hline U_l B_u & D_l \end{array}\right], T_2 = \left[\begin{array}{c|c} W_{rr}^H & W_{rl}^H V_{rl}^H \\ \hline U_{rl} B_{ru} & D_{rl} \end{array}\right] \tag{4.73}$$

$$T_3 = \left[\begin{array}{c|c} W_{rrr}^H & W_{rrl}^H V_{rrl}^H \\ \hline U_{rrl} B_{rru} & D_{rrl} \end{array}\right], T_4 = \left[\begin{array}{c|c} \cdot & V_{rrr}^H \\ \hline \cdot & D_{rrr} \end{array}\right] \tag{4.74}$$

Easy to see that the realization at k step is given by:

$$T_k = \left[\begin{array}{c|c} A_k & C_k \\ \hline B_k & D_k \end{array}\right] = \left[\begin{array}{c|c} W_r^H & W_l^H V_l^H \\ \hline U_l B_u & D_l \end{array}\right] \tag{4.75}$$

According to the reconfigured partitioning, we shall see that: for step k, on the current node, all right children belong to the further steps, while all left children go to the D_l in the realization of the current step. W_l, W_r and B_u are the translation matrices of the current node. U_l and V_l are column bases and row bases of the current node, yet they are not explicitly stored. Note that, according to the HSS definition, they should be generated(recursively) from the left children.

The conversion algorithm should start from the root node and proceed recursively. After constructing the realization on the current step, the algorithm proceeds by setting the right child as the current node and the algorithm goes recursively until it reaches the right bottom where no more right child exist. Then the realization of the last step will be given as:

$$\left[\begin{array}{c|c} \cdot & V^H \\ \hline \cdot & D \end{array}\right] \tag{4.76}$$

since a leaf does not have right child.

To show how a HSS tree is split as time-varying steps, we shall show the partition on a HSS binary tree in figure 4.5.

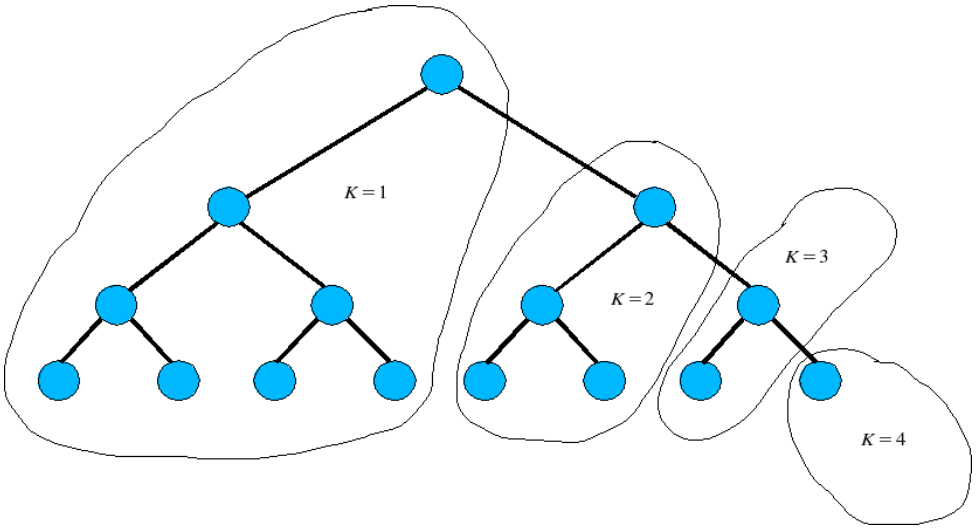

Figure 4.5: Binary tree partitioning

Now let's take a look at the realization formula, we see that D_l is a HSS block which can be a large HSS block. Another level of time-varying notation can be used to represent this D_l whose realization

may again contain sub-blocks represented by time-varying notation. Since U_l, V_l are not explicitly stored and can be derived locally from the current step, no efficiency is lost by applying recursive time-varying notation.

A number of remarks on recursive time-varying notation for HSS:

1. D_l in the realization is a HSS block which can either be represented by HSS representation or by time-varying notation. This implies a possible hybrid notation consisted of HSS representation and recursive time-varying notation.

2. U_l and V_l are HSS bases generated from D_l. For this recursive time-varying notation, they should not be explicitly stored and can be derived locally in the current step.

3. It is possible to represent general HSS matrices(not just block upper-triangular matrices) with this recursive time-varying notation.

4. All fast HSS algorithms can be interpreted in a recursive time-varying fashion.

5. Some algorithms applied on time-varying notation described in [9] can be extended to this recursive time-varying notation (HSS representation).

6. Moore-penrose solver applied on time-varying notation can be extended to this recursive time-varying notation.

Chapter 5

Iterative algorithms with Hierarchically semi-separable representations

In this chapter, we will present various iterative solution algorithms based on HSS representation. Firstly, in section 5.1, the motivation of iterative methods is presented; while in section 5.2, stationary iterative methods like Jacobi and SSOR iterative methods are described. In the section 5.3, more advanced CG like methods are discussed. Numerical experiment results based on OCAML implementation are given.

5.1 motivation of iterative solution method based on HSS representation

Matrix operation algorithms based on HSS representation is quite interesting in its own right. Much of its research has been triggered by a problem that can be posed simply as: Given $A \in C^{m \times n}, b \in C^m$, find solution vector(s) $x \in C^n$ such that $Ax = b$. (If the system A is overdetermined or underdetermined, the moore-penrose solution will have to be computed.) Many scientific problems lead to the requirement to solve linear systems of equations or multiply a system matrix with a vector as part of computations. Thus, the efficient fast HSS solution algorithms as well as the fast matrix-vector multiplication algorithm are quite useful, given the computation can be done efficiently.

Unfortunately, matrix operation algorithms based on HSS representation are not always efficient, they are only so when the rank of the off-diagonal sub-matrices are small. In chapter 4, the computation complexity analysis has been done with the assumption that the rank of the off-diagonal sub-matrices are neglectable (say around 10); while in practice, the system matrix of 2D or 3D problems does not fit in this categorial. To reduce the rank of the off-diagonal sub-matrices, the rows and columns of the system matrix shall be reordered. Graph theory has been of great help in re-arranging the non-zero entries of sparse matrices to reduce the *fill-in*[1] s in the factor matrix L and U (in case of direct solution method) or to move as many non-zeros as possible onto the diagonal to reduce the number of iterations needed (in case of iterative solution method). The most commonly used heuristic for performing reordering is the minimum degree algorithm [13, 32],An alternative approach, nested dissection ordering [12], has many appealing theoretical properties, but building an implementation that gives comparable ordering qualities and run times to the minimum degree method has proven to be very difficult. Some promising results have been demonstrated [3, 1] ...etc.

Unfortunately none of these methods has produced consistently better ordering than the minimum degree reordering method, and all require significantly amount of run time [19]. What is worse, even

[1] where the fill-in is the set of zero entries in A that become nonzero in L

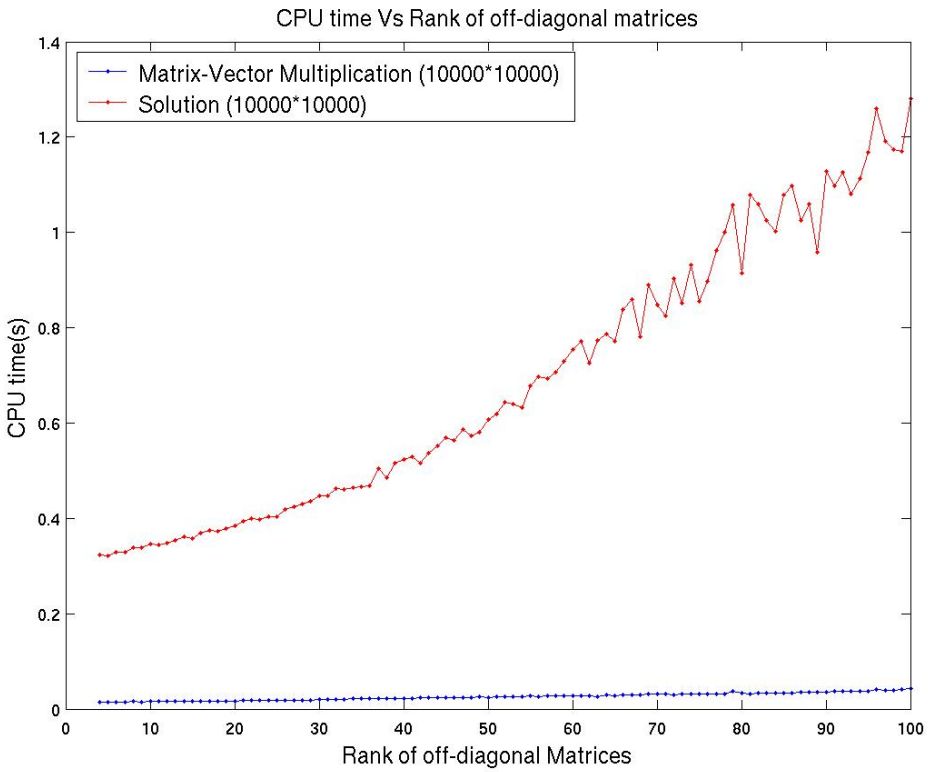

Figure 5.1: Computation time of Matrix-vector multiplication and solution Vs the rank of off-diagonal matrices

with these methods, the rank of the off-diagonal sub-matrices is not as small as expected. For simplicity, we assume that the rank of off-diagonal sub-matrices is less or equal to k then the upper bound of the computation complexity would be $O(N \times k^2)$ where N is the dimension of the system matrix. For direct solving algorithm, the computation complexity would be approximately of $O(N \times k^3)$. To illustrate how the size of the translation matrices would affect the computation time of Matrix-vector multiplication as well as the solution algorithm, we shall plot the computation time of these algorithms on matrices of the same size but with translation matrices of different size. To compare the computation time needed by solution and matrix-vector multiplication, we plot them together on a single figure 5.1. (Note that the above experiment is done on 10000×10000 matrices with the rank of off-diagonal matrices equal to a constant k.)

Again for comparison, the rate of the computation time needed by both algorithms is plotted as in figure 5.2. A few observations can be made based on the above figures:

1. From the figure 5.1, it is quite obvious that solution takes much more time than multiplication.
2. Again from the figure 5.1, The computation time of both algorithms increase dramatically with the rank constant k.
3. From the figure 5.2, the rate of the computation time of solution over that of matrix-vector multiplication increase linearly (approximately) with the rank k of the off-diagonal matrices, which is consistent with the theoretical conclusion.

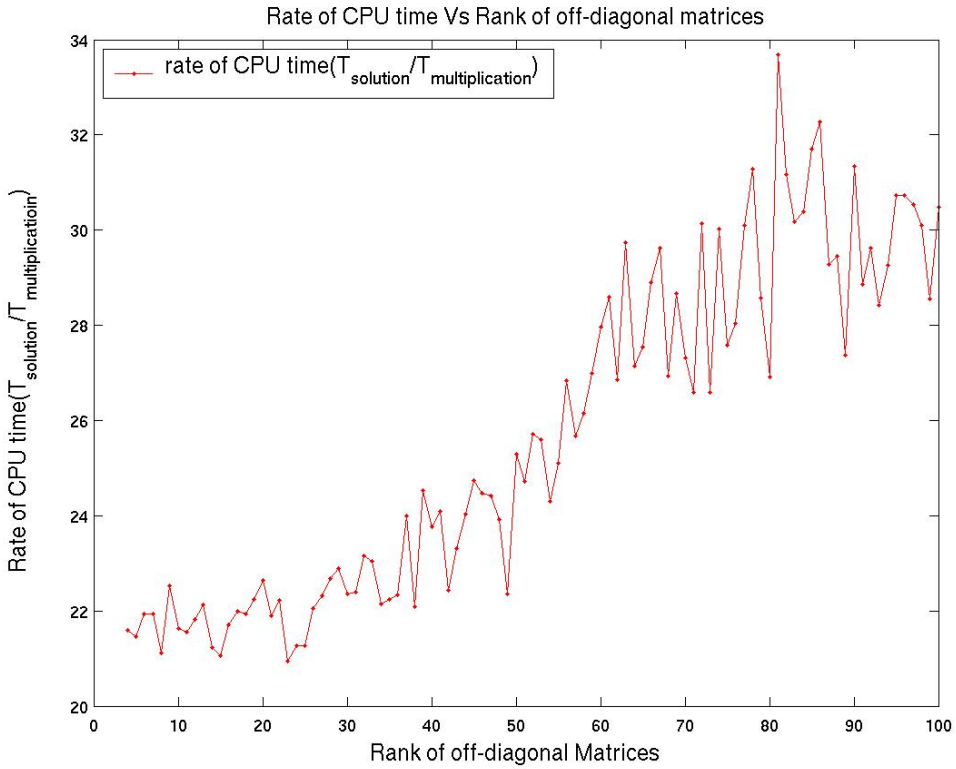

Figure 5.2: The rate of the computation time of Matrix-vector multiplication and solution Vs the rank of off-diagonal matrices

All these make it meaningful to develop iterative solution methods where the *approximate* solution is improved iteratively. For each iteration, matrix-vector multiplication is the crucial operation. If the number of iterations needed is sufficiently small, the iterative solution methods should take less time than the direct HSS solution method does. Another benefit is that, in our application, we will have to solve a system of equations in time domain, where the solution of one moment does not differ too much from the solution of the very last moment. This indicates that the solution of the last moment can be used as a good initial guess for the solution of this moment, the number of iterations can thus be reduced. In practice, it is not necessary to get a very accurate result (we make mistake during discretization anyway, so why should the solution method be so accurate?). In our case, a few percent of error is tolerable. This helps us to further reduce the number of iterations needed. To summarize, all these above give us a good motivation to study iterative solution methods based on HSS representation.

5.2 Stationary algorithms

A basic iterative method has the form

$$Cd^{l+1} = -r^l, x^{l+1} = x^l + d^{l+1}, l = 0, 1, 2, ... \tag{5.1}$$

where r^l is the residual of the lth iteration, d^{l+1} is the correction at lth iteration. x^0 is the initial approximation, sometimes chosen arbitrarily, or can be chosen to be $x^0 = C^{-1}b$. where C is nonsingular; C is what we call the *preconditioning* matrix or *preconditioner* for short. A good preconditioner can accelerate convergence a lot. There exist various ways to choose preconditioner; at the end of this chapter we will propose a number of fast algorithms to compute *preconditioners* , which are suitable for the HSS representation.

It can be shown that the basic iterative method (5.1) can be generalized and improved by introducing iteration parameters. For matrices with real and positive eigenvalues, parameterized iterative methods, (no matter of first order or of second order), will converge for any initial vector if only the parameters are taken to be sufficiently small. And it is true that the choice of the initial vector as well as the iteration parameters affect the convergence rate a lot.

Basic iterative methods are normally taken to be of first-order or of second-order, the first-order method is defined below, while the second-order method is defined similarly. People can, of course, derive third order or forth order iterative methods, but that does not pay. If appropriately parameterized, the convergence rate of the second-order iterative method is improved by an order of magnitude over that of the first-order iterative method [2].

Definition 3 *A* first-order, or one-step iterative, solution method is defined by

$$Cd^{l+1} = -\tau_l r^l, x^{l+1} = x^l + d^{l+1}, l = 0, 1, 3, ... \tag{5.2}$$

where $r^l = Ax^l - b$,τ_l is a sequence of parameters and x^0 is given. If $\tau_l = \tau, l = 0, 1, 2, ...$, then the method is called *stationary* ; otherwise *nonstationary* .

Due to the low convergence of the basic iterative solution method, these methods are not so popular, except some of them may be used as *preconditioners* (for instance SSOR and Jacobi). Therefore, we will only go though a few examples of the first-order stationary iterative methods to appreciate the originate of iterative solution methods.

5.2.1 (Block) Jacobi algorithm

The (block) Jacobi method(Gauss, 1823, and Jacobi 1845) or the simultaneous iteration method is probably the oldest iterative solution method. This idea goes as follows:

Definition 4 *Solution problem: Given a vector b and a nonsingular matrix A, find a vector x such that $Ax = b$ holds.*

Given the problem defined above, first, split A as $A = D_A - B_A$ where D_A is the diagonal part of A, More generally, A can be partitioned into block matrix form:

$$A = \begin{bmatrix} D_1 & A_{1,2} & ... & A_{1,n} \\ A_{2,1} & D_2 & ... & A_{2,n} \\ . & & & \\ . & & & \\ . & & & \\ A_{n,1} & A_{n,2} & ... & D_n \end{bmatrix} \tag{5.3}$$

Here, let $D_A = \text{diag}(D_1, D_2, ..., D_n)$ is the block diagonal part of A. Then the iterative update formula of Jacobi algorithm can be posed in a matrix operation form as follows:

$$D_A x^{l+1} = B_A x^l + b \tag{5.4}$$

or equivalently

$$x^{l+1} = D_A^{-1} B_A x^l + D_A^{-1} b \tag{5.5}$$

This method is quite easy to understand and implement, but convergence is not satisfactory. In fact, it does not converge for a large number of system matrices. The convergence is guaranteed when the system matrix is highly diagonal dominated;

Definition 5 *The matrix A is said to be strictly* diagonal dominant if $|a_{ii}| > \sigma_i, i = 1, 2, ..., n$, where $\sigma_i = \sum_{j=1, j \neq i}^{n} |a_{ij}|$.

Even then, the method may break down in case of a zero pivot(for scalar Jacobi version) or a singular D_i diagonal block(for block Jacobi version) is encountered.

numerical result of Block Jacobi method based on HSS representation

To study the convergence behavior of the Block Jacobi method based on HSS representation, we experiment this method with the test system matrices downloaded from MatrixMarket[2]; Those system matrices come from real scientific applications; and normally the detailed information of these matrices are given. We run the algorithm with these matrices and study the convergence behavior.

It turns out that: the Block Jacobi method only converges for less than half of these system matrices, especially for the matrices that are really highly diagonal dominated.

To study the convergence rate of this method, we experiment this method on smooth matrices A defined below, of which diagonal dominance is guaranteed.

$$A_{ij} = \begin{cases} c \times d & i = j \\ \frac{d}{|i-j|} & i \neq j \end{cases} \tag{5.6}$$

[2]http://math.nist.gov/MatrixMarket/

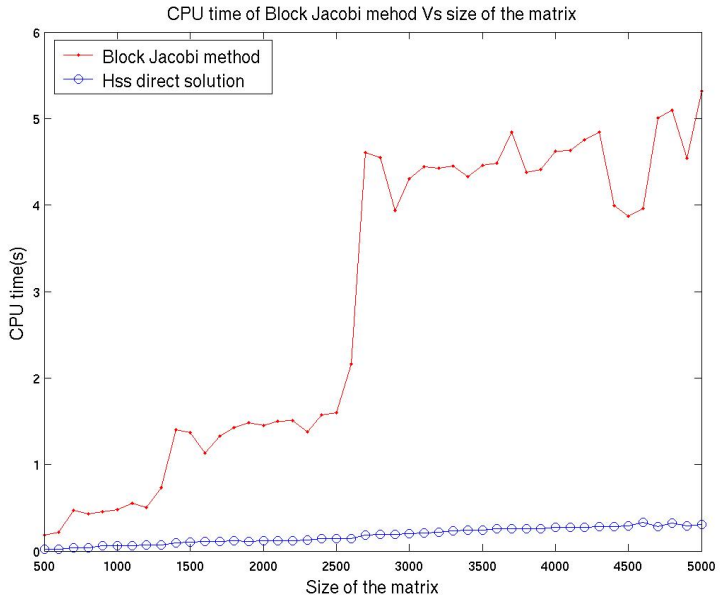

Figure 5.3: The CPU time needed by Block Jacobi method with HSS algorithms Vs size of the matrix

(Here d is the dimension of the matrix, c is a parameter to control the diagonal dominance)

To make sure the matrix generated is highly diagonal dominated, we choose the value of c to be 10 for the experiments on Block Jacobi method (HSS version). It is important to know that Block Jacobi method only converges for these highly diagonally dominated matrices. Hence this method is not applicable to most of system matrices.

We shall plot the CPU time needed to get 0.001% solution accuracy Vs the size of system matrices as follows (the CPU time needed by the direct HSS solver for the same set of matrices is also plotted for comparison):

Definition 6 *Solution accuracy: since in iterative methods, the exact solution is not available, the accuracy of the approximation solution is not known. Thus, we define the accuracy as the frobenius norm of the residual r over that of the righthand side b.*

Although not plotted in the figures, it is notable that the number of iterations needed by Block Jacobi method(HSS version) depends a lot on the diagonal dominance. When the matrix is *strictly diagonal dominated* . The iterative method converges in a few iterations (say around 100).

Figure 5.3 shows that the CPU time of solving a *strictly dominated* smooth matrix using this method increases quite irregularly with the size of the system matrix (actually, this is a problem for most of the iterative methods. That is the computation time needed is normally unpredictable). For those without diagonal dominance, the method converges quite slow or even diverges. If the method converges within a small number of iterations, the CPU time needed is comparable with that of the direct HSS solver. But its time curve is quite irregular, which is normally the case for iterative solution methods. Although not shown in these figures, it is mentionable that the direct solver is much more stable than iterative methods.

With all its limitations, the Block Jacobi method is only useful as *preconditioners*.

5.2.2 The (Block)Successive Overrelaxation algorithm based on HSS representation

The Successive Overrelaxation method, or SOR, is derived by applying extrapolation to Gauss-Seidel method. The extrapolation takes a weighted average between the solution of the previous iteration and the computed Gauss-Seidel solution of this iteration.

It is easier to explain this method in matrix computation form. Suppose a matrix A. Split A as follows:

$$A = L + D + U \tag{5.7}$$

where L is lower block triangular matrix, and U is upper block triangular matrix. In each iteration, the following update is computed:

$$x^k = (D - \omega L)^{-1}(\omega U + (1 - \omega)D)x^{k-1} + \omega(D - \omega L)^{-1}b \tag{5.8}$$

or equivalently:

$$x^k = (D - \omega L)^{-1}[(\omega U + (1 - \omega)D)x^{k-1} + \omega b] \tag{5.9}$$

Here, ω is the extrapolation factor which is used to accelerate the rate of convergence. While its value has to be chosen in the interval $(0, 2)$; outside it, the method won't converge. With the spectral radius ρ of the block Jacobi matrix is known, the theoretically optimal value of ω can be computed by:

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho^2}} \tag{5.10}$$

However, the spectral information is normally not known in advance, and computing it is prohibitively expensive. In order to fix a optimal ω for a special kind of system matrix, we can first experiment the method with a small system matrix and find the best ω for it. If one can be really naive, this ω can be used as the optimal one; this choice is open to doubt. The inverse of a block triangular HSS matrix can be applied on a vector easily via the forward and backward substitution method described in [23].

5.3 Krylov Space iterative algorithms

The Krylov space iterative algorithm derives its name from the fact that it minimizes certain residual functional over certain vector spaces called Krylov spaces.

Definition 7 *Krylov spaces: Given a nonsingular $A \in C^{N\times N}$ and $y \neq 0 \in C^N$, the nth Krylov (sub)space $K_n(A, y)$ generated by A from y is $K_n = K_n(A, y) = span(y, Ay, ..., A^{n-1}y)$.*

In this thesis, we will only study CG like methods, since this kind of solution methods have been quite popular and efficient. Other types of Krylov spaces method include Lanczos-Type methods.

5.3.1 Standard Conjugate Gradient method with HSS representation

The Conjugate Gradient method or CG method is an efficient method for *symmetric positive definite* systems. It is probably the oldest and best known non-stationary iterative method[3]

To derive the CG method, it is important to understand the following theorem:

Theorem 2 *If A is symmetric positive definite, the quadratic function $f(x) = \frac{1}{2}x^T Ax - b^T x$ is minimized by the solution of $Ax = b$.*

The CG method iteratively minimizes the quadratic function with x chosen from the *Krylov space* span the residuals $r^{(0)}, r^{(1)}, ..., r^{(l)}$. An article by Jonathan Richard Shewchuk [31] gives an comprehensive and interesting introduction to CG method.

Because of the popularity of the CG method, there are a large number of literatures on every aspect of this method, including error analysis, convergence analysis, and so on. Here we only study the time curve of the CG method combined with our HSS solver and matrix-vector multiplication method. The pseudo-code of the CG method combined with HSS algorithms can be found in A.1. Another important thing for CG method is that, due to the round-off error, the sequence of residuals may lose orthogonality after a certain number of iterations. So a CG method without any preconditioner may not converge within d iterations(d is the dimension of the matrix); while the CG method with exact arithmetic is guaranteed to converge with d iterations. That is why the CG method was once considered as a direct solution method. In practice, the CG method is almost always used with certain preconditioners.

Numerical results

Here, we only study the HSS CG method with Block Jacobi preconditioner. More about preconditioners are discussed in section 5.4.

For the HSS algorithms to be efficient, we choose to do the experiment on smooth matrices defined as:

$$A_{ij} = \begin{cases} c \times d & i = j \\ \frac{d}{|i-j|} & i \neq j \end{cases} \tag{5.11}$$

(d is the dimension of the matrix, c is a parameter to control the diagonal dominance. For the experiment with CG like methods, we choose the value of c to be 2, since diagonal dominance is not so important for the CG like methods.) The *solution accuracy* is specified to be 10^{-6}; the configuration is the same for the experiments with other CG like methods.

In the figure 5.4, the CPU time needed by direct HSS solver on the same set of matrices is also plotted. It can be seen from the figure that the time curve of the CG method is much less regular than that of the direct solver. In fact, it is well understood that the convergence behavior of the CG method depends on some miserddable spectral properties of the system matrix. It is also notable that in this experiment, the CG method with HSS algorithms takes more time than HSS direct solver. This is however not the case if the system matrix is less smooth. We shall study how the smoothness of the system matrix affects the trade-off between CG like method and HSS direct method at the end of this chapter.

[3] non-stationary methods differ from the stationary methods in that computations involve information gathered at each iteration.

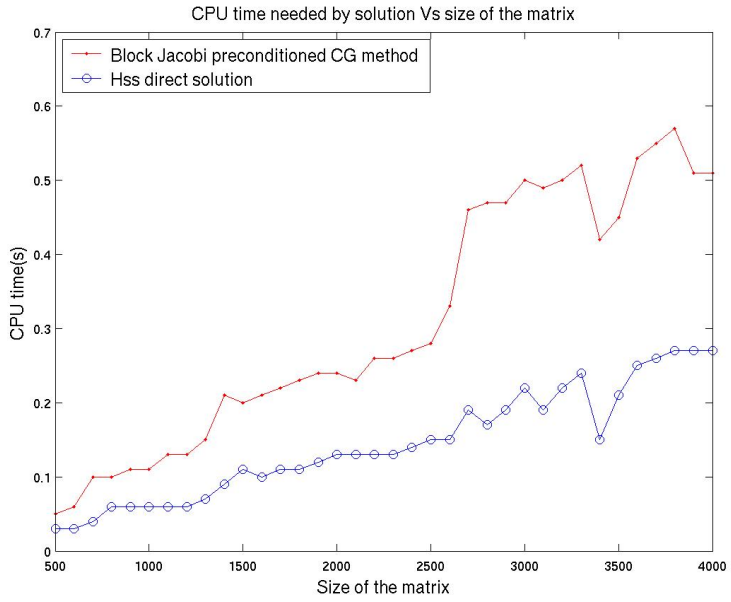

Figure 5.4: The CPU time needed by the diagonal preconditioned CG method with HSS algorithms Vs size of the matrix

5.3.2 BiCG method with HSS representation

As mentioned above, the CG method is not suitable for most of the non-symmetric matrix because the residual vector can not be made orthogonal with short recurrences (for a proof, refer to [36]), we can also use other Krylov space methods like GMRES method to retain orthogonality of the residuals by long recurrences. However, in those methods, even more storage is needed. BiCG method remedies this problem with another approach. It replaces the orthogonal sequences by two mutually orthogonal sequences. This method sacrifices some computation time to compute the transpose of A; and in each iteration, above twice much computation is needed compared to the CG method. The pseudo-code of this BiCG method combined with Hss algorithm can be found in appendix A.2.

Numerical results

Here we just choose to study the time curve of the BiCG method. Similar to the analysis of CG method, the same set of experiments as that in section 5.3.1 is done on BiCG method. The computation time Vs the size of matrix is plotted in figure 5.5.

It can be seen that the CPU time needed by BiCG method is more than that needed by CG method to solve the same set of system matrices. That is the price we pay to improve the applicability of the CG method. Note that, although not plotted here, the BiCG method needs about twice as much storage as needed by the CG method (to store the HSS representation of the transpose of the original system matrix). Other than these above, the convergence behavior of BiCG method does not differ too much from that of the CG method.

In practice, the convergence behavior of BiCG method may be quite irregular, and the method may even break down. Look-ahead strategies may be used to remedy this problem, but we are not going to go though these in this thesis.

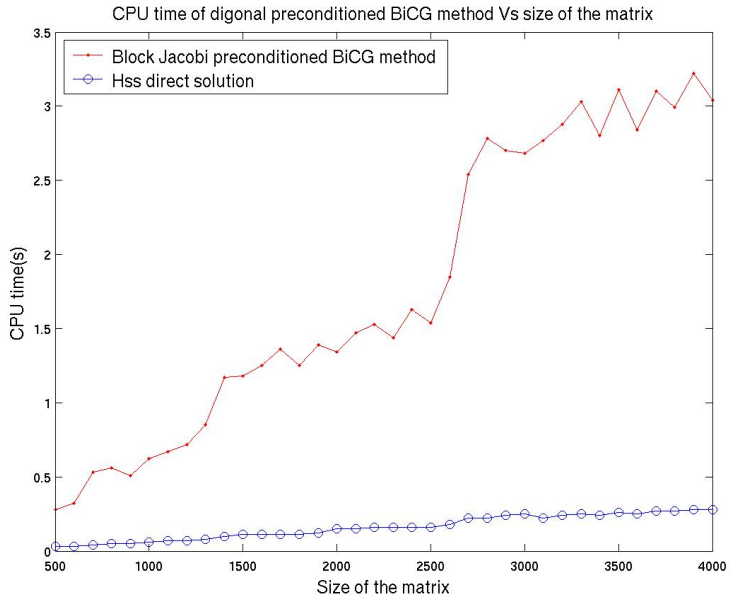

Figure 5.5: The CPU time needed by the diagonal preconditioned BiCG method with HSS algorithms Vs size of the matrix

5.3.3 CGS method with HSS representation

As mentioned in the last subsection, BiCG method needs more storage for the transpose of the system matrix A; also in each iteration, two matrix-vector multiplications are needed. To improve the efficiency of this method while keep its applicability, several variants (CGS and Bi-CGSTAB) have been proposed to improve the method in certain circumstance. The CGS method is proposed to increase the convergence and save on storage, while the Bi-CGSTAB method is proposed to improve the stability of the BiCG method in case of breakdown situation. However it is observed that the Bi-CGSTAB is not as stable as its name tells. In this section we will study the time curve of the diagonally preconditioned BiCG method. The pseudo-code of the (preconditioned) CGS method combined with HSS algorithms can be found in appendix A.3.

Numerical results

Similar experiment as that in section 5.3.1 is done on this method in order to plotted the computation time Vs the size of the HSS matrices in figure 5.6:

Often one would expect the convergence of CGS method is about twice as fast as for BiCG method (on the same system matrices, with the same preconditioner).

5.3.4 Summery on the three CG like methods compared to HSS direct method

To summarize the three CG like methods above and to show the benefits we get from developing these HSS iterative algorithms. We shall put the time curves together in one figure as below (note here, due to the OCAML implementation of the HSS direct solver, after solution with HSS direct solver, the original HSS tree would be destroyed; in order to keep the system intact, a copy of the original system should be

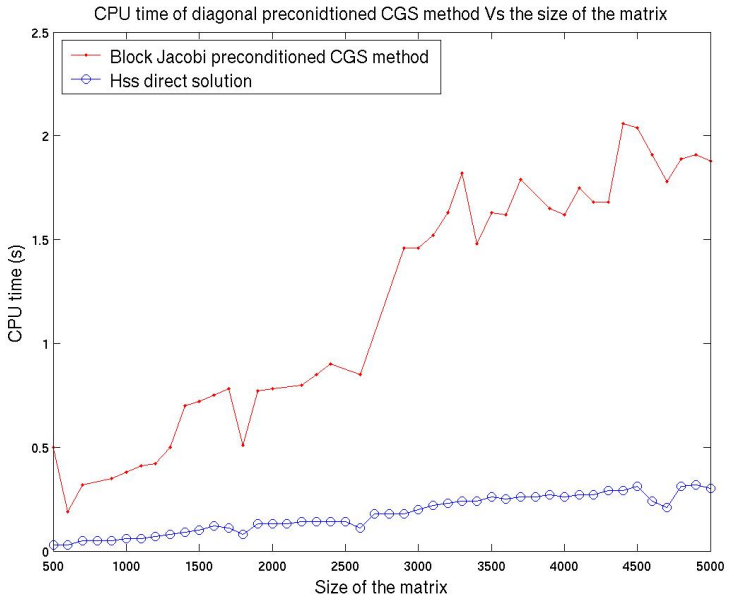

Figure 5.6: The CPU time needed by the diagonal preconditioned CGS method with HSS algorithms Vs size of the matrix

made and the direct solver would work on the copy. The time spent on copying the HSS tree should also be taken into account when analyzing the performance of the direct solver):

It can be seen that from the figure 5.7 that the CPU time needed by the HSS CG method is comparable with that of the direct solver (they are within the same order). Among the CG like methods, standard CG method takes the least time, however, its applicability is not as good as that of others. The HSS CGS method takes about half of the time (normally more than 50%) needed by HSS BiCG method; this is consistent with the analysis of these two methods. It can also be seen from the figure that the time curve of iterative methods are quite irregular, while the direct solver scales well with the size of the matrices (if the system matrices are smooth on off-diagonal sub-matrices).

After these analysis above, one question still remains: under what situation should the iterative methods be preferred over the direct solver? As we mentioned in section 5.1, the core operation of iterative methods is matrix-vector multiplication; this operation as we have shown scales better with the *HSS complexity* than the direct HSS solution method does. This indicates that the iterative methods should be adapted under the circumstance that the off-diagonal sub-matrices of the HSS matrices is not of low rank (That is when the *HSS complexity* is large compared to the size of the matrices.). We shall do a series of experiments to see how the iterative methods and direct method would scale with the smoothness.

We choose to work on the following matrix A:

$$A_{ij} = \begin{cases} c \times d & i = j \\ d \times \cos(k|i-j|\pi) & i \neq j \end{cases} \tag{5.12}$$

Here, k is used to control the smoothness, a larger k would result in more high frequency components, which would then result in less smooth matrices. d is the dimension of the matrix. d here is specified as 2000; that is the matrices are of size 2000×2000. c controls diagonal dominance; we choose the value of c be 1.25 in this experiment. We shall plot the CPU time needed to get 10^{-6} *solution accuracy* by

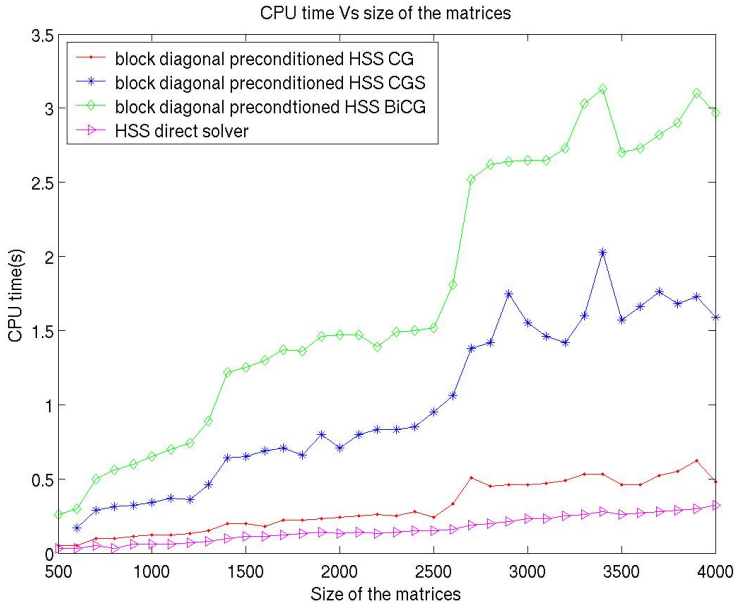

Figure 5.7: The CPU time needed by the diagonal preconditioned CG like methods with HSS algorithms and direct HSS solver Vs size of the matrix

diagonally preconditioned HSS CG[4]and direct HSS solver Vs the value of k. We shall also plot the CPU time needed by both algorithms Vs the *HSS complexity* , to show how the two algorithms would scale with the increasing *HSS complexity* .

From the figure 5.8, it is quite obvious that the direct solution method does not scale well with the increasing value of k. While the CPU time needed by CG method scales linearly with the value of k. The increasing value of k would make the matrix less smooth, thus increases the *HSS complexity* of the HSS representation. It makes sense to plot the CPU time needed Vs the *HSS complexity* of the HSS representations in figure 5.9.

After the above comparison, it is safe to conclude that HSS iterative method should be preferred over direct HSS solution method, if the *HSS complexity* of the HSS representation is not small compared to the dimension of the matrix. Or equivalently, the iterative method should be preferred when the matrix is not very smooth. However if the matrix is completely not smooth, the HSS algorithms described in this thesis are not recommended. All these put us in a trade-off box.

5.4 HSS Proconditioner construction

As well studied by other researchers, the convergence rate of various iterative methods depends on spectral properties of the coefficient matrix. Thus, the system matrix can be transformed into an equivalent one in the sense that it has the same solution, but has more favorable spectral properties. A *preconditioner* is the matrix that effects such transformation[27].

A *preconditioner* is in fact an approximation to the original system matrix A; In order to gain any speedup, this *preconditioner* should be easy to compute and the inverse of this approximation matrix

[4]More advanced preconditioners can be found in section 5.4.

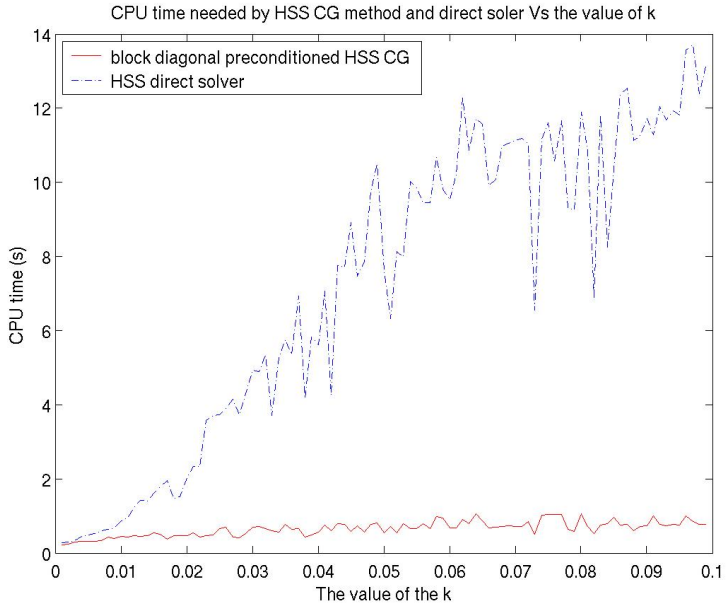

Figure 5.8: The CPU time needed by the diagonal preconditioned CG method with HSS algorithms and direct HSS solver Vs the value of k

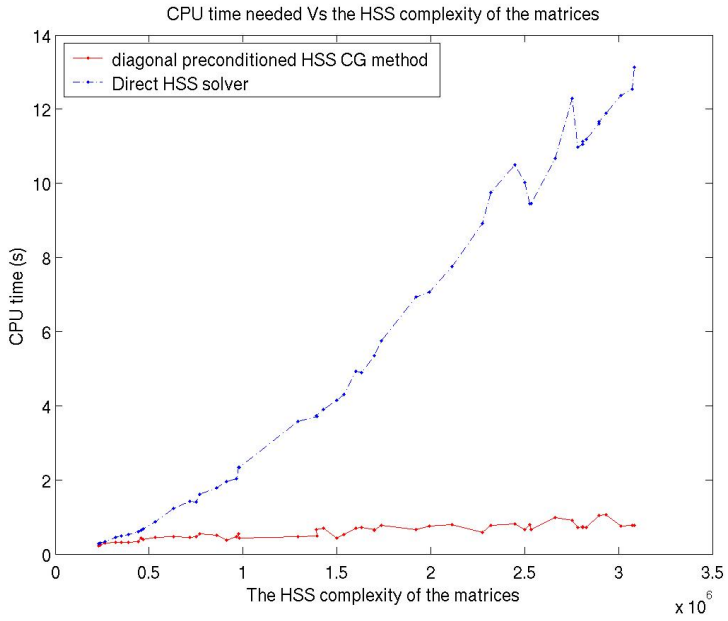

Figure 5.9: The CPU time needed by the diagonal preconditioned CG method with HSS algorithms and direct HSS solver Vs the *HSS complexity* of the HSS matrices

should be easy to apply on any vector. For instance, given the problem defined in definition (4), suppose the *left preconditioner* M approximates A in some way, the transformed system would be:

$$M^{-1}Ax = M^{-1}b \tag{5.13}$$

If M is nonsingular, the solution of the transformed system is identical with the original system, but the spectral properties of $M^{-1}A$ may be more favorable. Also note that, Neither M^{-1} nor $M^{-1}A$ is explicitly computed, the inverse of M is applied to a vector x by solving a linear system with coefficient matrix M and righthand side x. An ideal *preconditioner* should be easy to construct, easy to solve and improve the convergence rate of iterative methods as much as possible (by improve the spectral properties of the coefficient matrix). Unfortunately, the these three nice properties are normally against each other.

Therefore, there is a trade-off between the cost of constructing such *preconditioner* , and the gain in convergence rate. Some *preconditioners* aim for low cost, others aim for effectiveness. Here we propose a few *preconditioners* , of which the OCAML implementation is available, to accelerate the convergence rate.

5.4.1 Block Diagonal preconditioner of HSS representation

Given the problem defined in definition 4, with the assumption that A is given in its HSS representation. The simplest preconditiner M consists of just the diagonal of the HSS matrix A.

$$M = \mathbf{D} \tag{5.14}$$

where $\mathbf{D}$ is the on-diagonal sub-matrix of the HSS matrix A defined in equation 1.12. This is known as block Jacobi preconditioner.

The inverse of this block diagonal matrix M can be computed by inverting the matrix block-wise. Another benefit is that the preconditioner does not need memory, it shares the same data with the originate HSS representation; it is also quite easy to implement.

On the other hand, more sophisticated preconditioners usually yield a larger improvement. How the Block Diagonal preconditioner would approximate the original HSS matrix depends on how much the original matrix is diagonal dominated. With rather poor diagonal dominance, A block diagonal preconditioner does not improve the convergence rate a little bit.

5.4.2 SSOR preconditioning of HSS representation

Another 'cheap' preconditioner like Block Jacobi preconditioner is the SSOR preconditioner. Like Block Jacobi preconditioner, this preconditioner can be derived without any work and additional storage.

Suppose the original system A is symmetric, we shall decompose A as

$$A = D + L + L^T \tag{5.15}$$

where L is a block lower triangular matrix. Then the SSOR matrix is defined as

$$M = (D + L)D^{-1}(D + L)^T \tag{5.16}$$

usually, M is parameterized by ω as follows:

$$M(\omega) = \frac{1}{(2-\omega)}(\frac{1}{\omega}D + L)(\frac{1}{\omega}D)^{-1}(\frac{1}{\omega}D + L)^T \tag{5.17}$$

The optimal value of ω will reduce the number of iteration needed significantly. However, computing the value of the optimal ω needs the spectral information which is normally not available in advance and prohibitively expensive to compute. The way to apply such preconditioner on vector x has been discussed in section 5.2.2.

5.4.3 Fast model reduction with fast solver preconditioner

For the same problem and same assumption as in section 5.4.1, we can use model reduction method defined in section 3.1.3. While with different factorization methods, different model reduction methods can be developed, various OCAML routines are available with parameters to specify the tolerance. Take the fast model reduction method with two modified SVD decompositions in each step, we experiment this preconditioner generator on a 2000×2000 HSS matrix (the HSS matrix is generated as a smooth matrix defined in formula 5.11)). Unfortunately, the model reduction methods are heuristic algorithms, it does not necessarily produce the optimal HSS preconditioner subject to certain tolerance, although model reduction with larger tolerance does reduce the HSS complexity better than the one with small tolerance does.

to see how the resulted HSS preconditioner approximates the original HSS matrix. The relative error is chosen to be normF$(A - M)$/normF(A), where normF(B) computes the *frobenius* norm of the matrix B. We shall also plot the HSS complexity of the resulted HSS preconditioners subject to different tolerance, to see how the *HSS complexity* (definition 1) of the HSS representation is reduced.

Keeping in mind that the ideal HSS preconditioner approximates the original matrix and reduces the *HSS complexity* as much as possible; the trade-off point depends a lot on the characteristics of the original matrix. Normally, the model reduced HSS preconditioner approximates hierarchically semi-separable matrices better than other matrices. To find the best parameters for the preconditioner construction procedure, numerical experiment has to be done on the practical system matrices.

From the figure 5.10, it seems that the relative error increases linearly with the tolerance specified. This is ,however, only the case when the off-diagonal sub-matrices are smooth. For a less structured matrices, one would expect discontinuity in figure 5.10.

While from the figure 5.11, it seems that the HSS complexity does not decrease linearly with tolerance, but rather decreases in a curve, one would expect a flat tail of the curve. This means for any tolerance specified, there is a lower bound for the HSS complexity of the model reduced preconditioner. In fact, when a sufficiently large tolerance is specified, the model reduced preconditioner becomes the block Jacobi preconditioner. and the HSS complexity of the block Jacobi preconditioner is the lower bound of the HSS complexity of the model reduced preconditioner.

Numerical results

To study how this preconditioner would improve convergence, we choose to combine this preconditioner method with Hss BiCG method and HSS CGS method[5] to see how many iterations Hss preconditioned BiCG and CGS need when preconditioners generated with different tolerance are provided. We shall work on the HSS matrix of size 2000×2000 defined in formula 5.11 with c specified as 1.5, so that the iterative algorithms would converge slow enough for us to observe differences with preconditioners of different tolerance. A series of *preconditioners* are constructed with tolerance specified from 10^{-3} to 1. The time needed to construct the preconditioners subject to different tolerance should be plotted.

[5]Here, BiCG and CG method instead of CG method are chosen because the left preconditioned system won't be symmetric. The CG method is generally not applicable to this non-symmetric system. see further

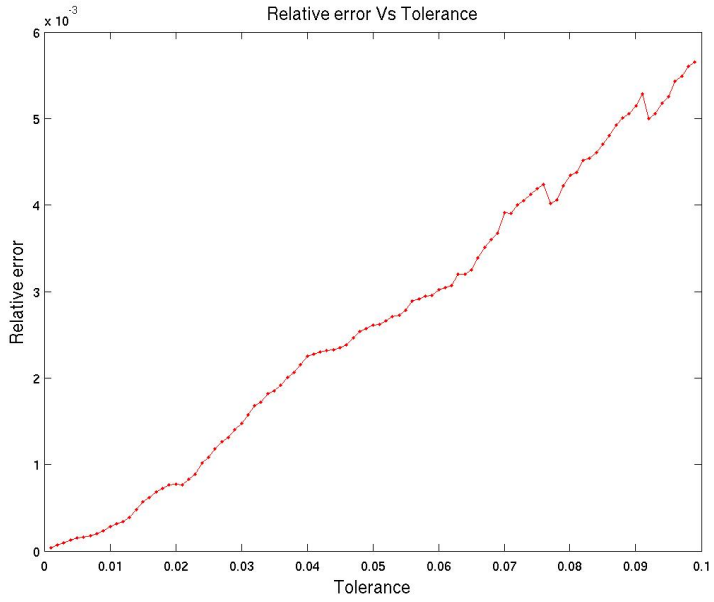

Figure 5.10: The relative error of the preconditioner Vs the tolerance specified to generate the preconditioner

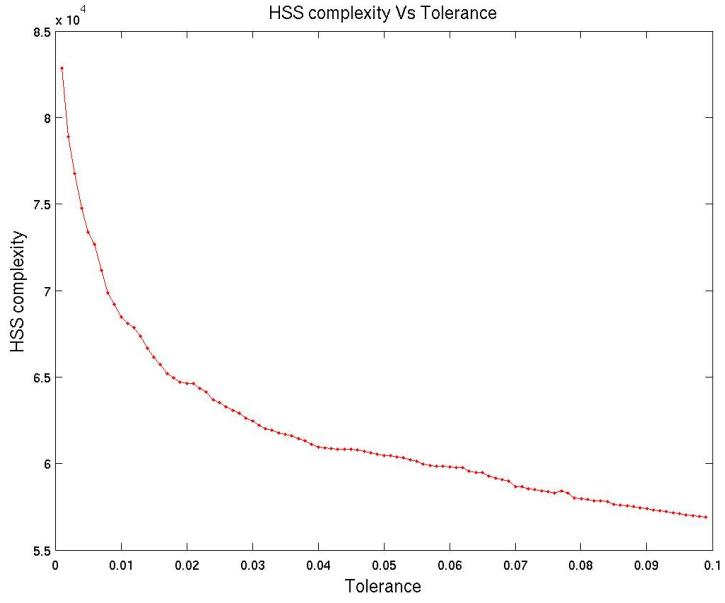

Figure 5.11: The HSS complexity of the preconditioner Vs the tolerance specified to generate the preconditioner

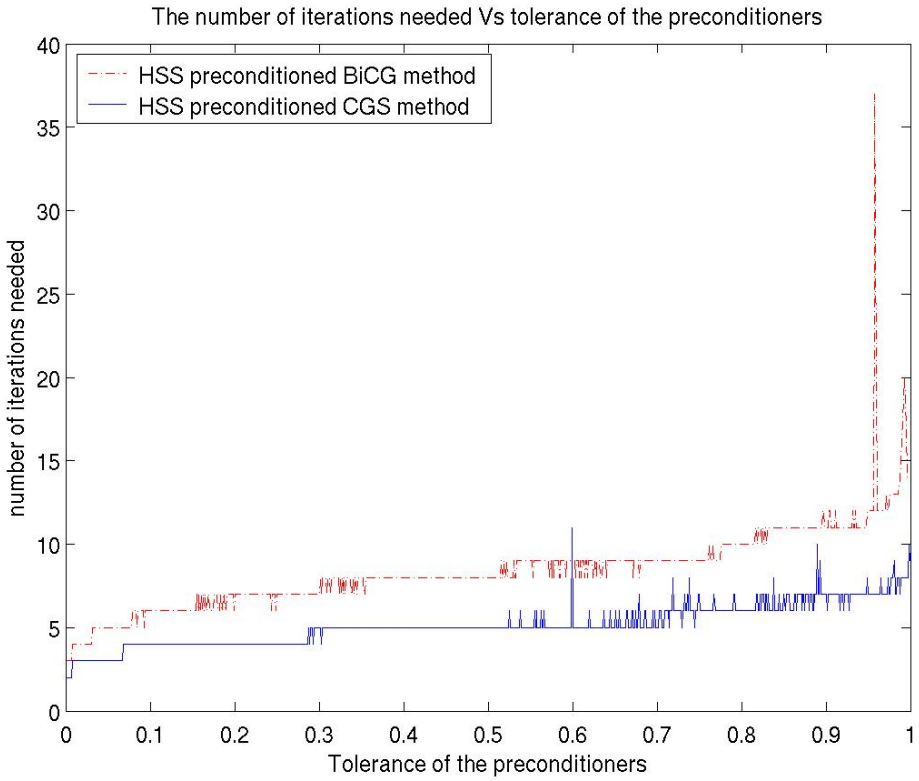

Figure 5.12: The number of iterations needed by the preconditioned iterative method to get 10^{-6} solution accuracy Vs the tolerance specified to generate the preconditioner

We then plot the number of iterations needed to get 10^{-6} *solution accuracy* by the preconditioned HSS BiCG method and preconditioned HSS CGS method Vs the tolerance of the preconditioners in figure 5.12.

The CPU time needed is plotted in figure 5.13. A few comments(observations) can be made on the figures (5.12, 5.13):

1. less CPU time is needed for constructing a preconditioner with larger tolerance.
2. a preconditioner with smaller tolerance results in less iterations (not necessarily less CPU time, since constructing and computing with a preconditioner with smaller tolerance needs more time.)
3. Under the same configuration, the preconditioned HSS CGS method converges about twice as fast as the preconditioned HSS BiCG method.

5.4.4 Fast model reduction with complete LU factorization preconditioner

It is mentioned in section 5.3.1 that: the standard CG method only works for symmetric positive defined matrix. If it is not the case, the standard CG method would converge quite slow or even diverge. we will of course expect the transformed system still be symmetric positive definite, if the original system is so.

Normally, the *left preconditioner* alone is often not what is used in practice. because, the transform matrix $M^{-1}A$ is generally not symmetric, even if A and M are symmetric. Then the CG method is not

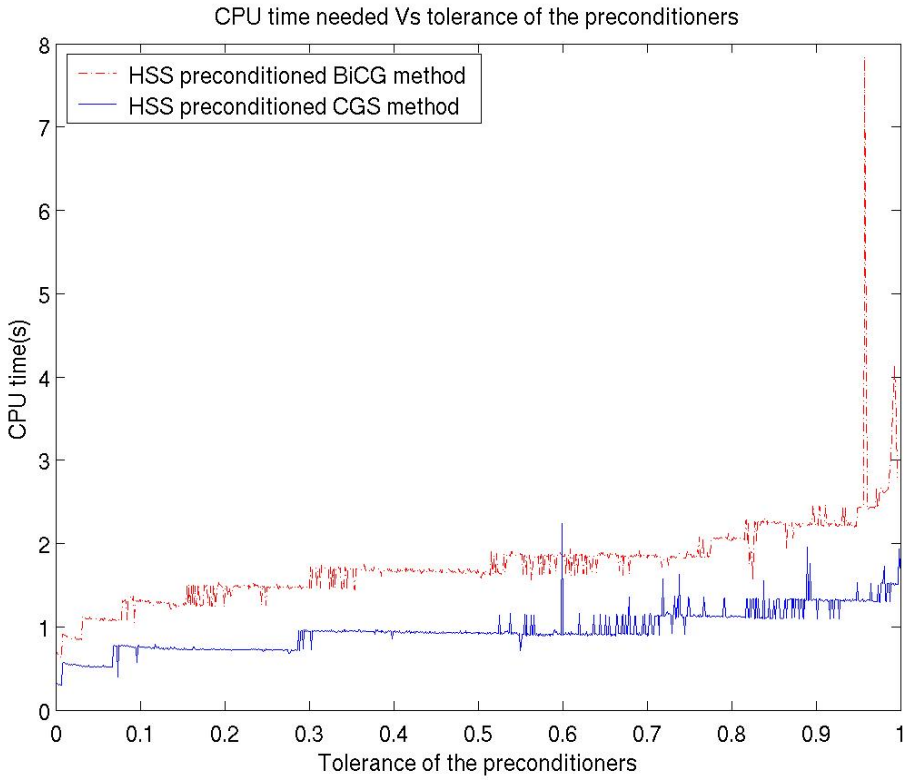

Figure 5.13: The CPU time needed by the preconditioned iterative method to get 10^{-6} solution accuracy Vs the tolerance specified to generate the preconditioner

immediately applicable to this system. We can of course use CGS and BICG method as described in section 5.3; as long as the original system is positive definite, the ideal of CG method is applicable again. However, people may still find many reasons to stick to the standard CG method with preconditioner. Firstly, CG method does not need transpose operation, thus save CPU time and memory in each iteration; it is mentionable that both CGS and BICG need more computation in each iteration than CG does. Secondly, if the original system is symmetric, it is of course advantageous to keep this property intact, so that only half of the system needs to be stored; it also means less computation, if one takes care of the redundant computation. Commercial numerical libraries (IMSL and NAG) are available to solve sparse symmetric systems efficiently. However a hierarchically semi-separable matrix does not have to be sparse; in fact, it is normally a dense matrix with smooth off-diagonal sub-matrices. Thus those commercial libraries are not useful for this kind of systems.

One way to remedy the preconditioner for standard CG method is to factorize the *left preconditioner* as $M = M_1 M_2$, and apply M_1 and M_2 separately as *left preconditioner* and *right preconditioner* . Then the original system would be transformed into the following:

$$M_1^{-1} A M_2^{-1} (M_2 x) = M_1^{-1} b \tag{5.18}$$

Here M_1 is called *left preconditioner* ; M_2 is called *right precondtioner* . If M is symmetric, that is $M_1 = M_2^t$ (note that if the original HSS matrix is symmetric, the *preconditioner* constructed by the algorithm presented in section 5.4.3 will be symmetric as well), one can easily prove that the transformed coefficient matrix $M_1^{-1} A M_2^{-1}$ is symmetric. Thus the standard CG method is applicable again. One only has to replace the initial residual r_0 as $r_0 := M_1^{-1} b$ before the iterative process and recover the solution vector x_n by letting $x_n := M_2^{-1} x_n$ after convergence is achieved.

M_1 and M_2 can be easily constructed by a combination of pre-existing algorithms as follows:

1. Firstly, apply the fast model reduction method described in section 5.4.3 with certain tolerance on the original system A to get the approximation matrix M.

2. Then, a HSS LU factorization method described in section 4.1.5 is applied on matrix M to get the lower triangular and upper triangular factor (M_1 and M_2 respectively). With the assumption that M is symmetric, the LU factorization described in section 4.1.5 becomes Cholesky decomposition where $M_1 = M_2^t$.

3. After M_1 and M_2 are ready, the iterative solver with double side *precondtioner* can be used to solve the system.

Since the LU factorization method is COMPLETE, the *relative error Vs tolerance* curve and the *HSS complexity Vs tolerance* curve are exactly the same as in figure 5.10 and 5.11. But more computation is needed for factorization and for applying the inverse of M_1 and M_2 to vectors. Since both M_1 and M_2 are block triangular systems, fast HSS forward substitution and backward substitution methods from [23] can be used. With the LU factors available, solution of the triangular systems by fast forward and backward substitution only accounts for 17% of the factorization time[23].

Numerical results

The only question left is: how much does this preconditioner construction method cost in terms of computation. We will study it with two set of numerical experiments. First we hold the rank of the off-diagonal sub-matrices as 40 and the tolerance as 0.0001; with the parameters specified above, the

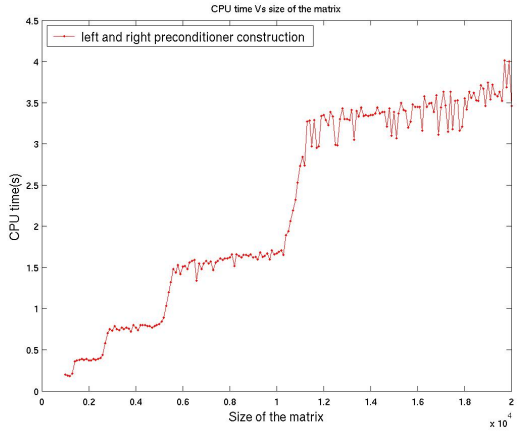

Figure 5.14: The CPU time needed to compute the double side preconditioner Vs size of the matrix

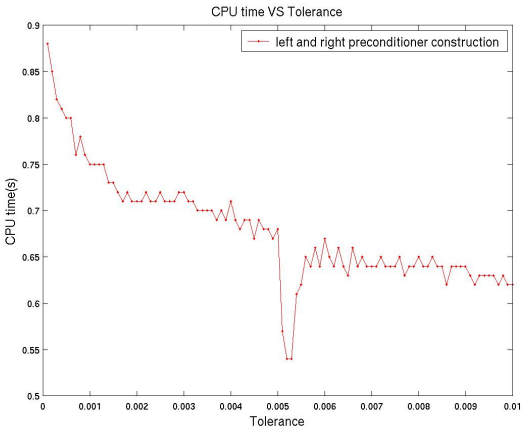

Figure 5.15: The CPU time needed to compute the double side preconditioner Vs the tolerance specified to generate the preconditioner

preconditioner construction method is executed many times on HSS matrices of different size so that the CPU time needed Vs the size of the matrices can be plotted in figure 5.14.

Then, we hold the size of the system matrix (which is a smooth matrix) as 2000×2000, and plot the CPU time needed to construct double side preconditioner Vs the tolerance in figure 5.15.

It is then easy to see that constructing the *preconditioners* with larger tolerance consumes less CPU time. The preconditioner described in this section should be at least as good as that the preconditioner in the last section (in terms of effectiveness). Here symmetrization is kept with computation time penalty.

Chapter 6

Integrated Field Equations method for Maxwell's Equations

In this chapter, the Space-Time Integrated Field Equation Method is adapted to compute full electromagnetic fields in simple 2D configuration. Point of departure are the space-time integrated equivalents of the first-order local EM field equations that couple the electric and magnetic field strengths, and the electric and magnetic flux densities to their generating source distributions. Together with the constitutive relations that represent the combined electric and magnetic properties, these are spatially and temporally discretized. The discretization technique is designed in such a manner that only the values of the continuous components of the EM fields occur in the computation [20]. The values of the discontinuous components are left free to jump across interfaces . Linear local edge expansions for the electric and magnetic field strengths and linear local face expansions for the electric and magnetic flux densities on a mesh (triangular mesh in 2D configuration) provide a consistent scheme for this purpose. Perfectly Matched Layers are used to truncate the computational domain. After properly assembling the local equations, a system of algebraic equations results that remains to be solved.

6.1 Method outline

In this section, we shall outline the procedure we will follow to produce the system matrix, solve the system, as well as post-processing procedure to compute the approximate quantities.

1. Triangular finite elements are used to discretized the computational domain in case of 2D configuration.

2. The systems matrix and the known right-hand side are assembled from the individual finite elements in the mesh.

3. A so called local method is used to eliminate the redundancy in the system, so that the system becomes a determined one which minimizes certain norm.

4. The system matrix shall be solved with certain solving technique

5. In the post-processing, The solution vector is then used to generate the desired numerical results. Among other things all approximated field quantities can be calculated and plotted.

6. The above procedure executing the system matrix construction procedure is executed iteratively in time steps so that all fields within the interesting time frame can be computed.

6.2 The Time Domain Integrated Field Equations

Before presenting the electromagnetic equations, the notations representing the electromagnetic fields quantities have to be defined:

$\vec{E}(t) =$ electric field strength $[V/m]$ at time t,
$\vec{H}(t) =$ magnetic field strength $[A/m]$ at time t,
$\vec{J}^{con}(t) =$ volume density of induced electric current $[A/m^2]$ at time t,
$\vec{K}^{con}(t) =$ volume density of induced magnetic current $[V/m^2]$ at time t,
$\vec{D}(t) =$ electric flux density $[C/m^2]$ at time t,
$\vec{B}(t) =$ magnetic flux density $[T]$ at time t,
$\vec{J}^{imp}(t) =$ volume source density of impressed electric current $[A/m^2]$ at time t,
$\vec{K}^{imp}(t) =$ volume source density of impressed magnetic current$[V/m^2]$ at time t

Now above field quantities are all vectorial quantities, it is also needed to represent one of its components in 3D space with the notation as follows:

$E_x(t)$ represents the x component of the electric field strength $\vec{E}$ at time t,
$H_y(t)$ represents the y component of the electric field strength $\vec{H}$ at time t
$etc...$

The total volume density of electric current and the total volume density of magnetic current are defined as:

$$\vec{J}(t) = \vec{J}^{con}(t) + \partial_t \vec{D}(t) + \vec{J}^{imp}(t) \tag{6.1}$$

$$\vec{K}(t) = \vec{K}^{con} + \partial_t \vec{B}(t) + \vec{K}^{imp}(t) \tag{6.2}$$

Given an arbitrary surface S and its boundary ∂S, the Maxwell equations can be represented in the integrated form as follows:

$$\oint_{x_r \in \partial S} \vec{H} dl = \int\int_{x_r \in S} \vec{J} ds \tag{6.3}$$

$$\oint_{x_r \in \partial S} \vec{E} dl = -\int\int_{x_r \in S} \vec{K} ds \tag{6.4}$$

With the compatibility relations in integrated form, given an arbitrary closed surface S', the compatibility relations are given by:

$$\oint\int_{x_r \in S'} \vec{J} ds = 0 \tag{6.5}$$

$$\oint\int_{x_r \in S'} \vec{K} ds = 0 \tag{6.6}$$

with the assumption that the media is instantaneously reacting, the constitutive relations are given by;

$$\vec{J}^{con}(x_r, t) = \bar{\sigma}^e(x_r, t)\vec{E}(x_r, t) \tag{6.7}$$

$$\vec{K}^{con}(x_r, t) = \bar{\sigma}^m(x_r, t)\vec{H}(x_r, t) \tag{6.8}$$

$$\vec{D}(x_r,t) = \bar{\varepsilon}(x_r,t)\vec{E}(x_r,t) \tag{6.9}$$

$$\vec{B}(x_r,t) = \bar{\mu}(x_r,t)\vec{H}(x_r,t) \tag{6.10}$$

where

$$\bar{\sigma}^e(x_r,t) = \text{electric conduction relaxation tensor } (S/ms) \tag{6.11}$$

$$\bar{\sigma}^m(x_r,t) = \text{magnetic conduction relaxation tensor } (/ms) \tag{6.12}$$

$$\bar{\varepsilon}(x_r,t) = \text{permittivity relaxation tensor } (F/ms) \tag{6.13}$$

$$\bar{\mu}(x_r,t) = \text{permeability relaxation tensor } (H/ms) \tag{6.14}$$

Here these tensors are given as 3×3 matrices which represent the constitutive relations. If the media is isotropic media, then all these 3×3 matrices are diagonal matrices. If the media is further isometric, the data entries are the same in each of these diagonal matrices.

If the media is isotropic, to access the these relaxation tensors in one direction. The following notation is defined:

$\sigma^e_{xx}(x_r,t) = \bar{\sigma}^e(x_r,t)_{[x,x]} =$ electric conduction relaxation tensor (S/m) in x direction
$\sigma^e_{yy}(x_r,t) = \bar{\sigma}^e(x_r,t)_{[y,y]} =$ electric conduction relaxation tensor (S/m) in y direction
$etc...$

6.3 2D Electromagnetic field

In this section, we are going to review the famous Maxwell equations for 2D Electromagnetic field. Since the 2D configuration is invariant in z direction. The material parameter functions and the source functions are independent of z coordinate; that is:

$$\bar{\sigma}^m = \bar{\sigma}^m(x,y,t), \bar{\sigma}^e = \bar{\sigma}^e(x,y,t), \bar{\varepsilon} = \bar{\varepsilon}(x,y,t), \bar{\mu} = \bar{\mu}(x,y,t) \tag{6.15}$$

$$\vec{J}^{imp} = \vec{J}^{imp}(x,y,t), \vec{K}^{imp} = \vec{K}^{imp}(x,y,t) \tag{6.16}$$

It is also well understood that the electromagnetic field is independent of z direction.

$$\vec{E} = \vec{E}(x,y,t), \vec{H} = \vec{H}(x,y,t) \tag{6.17}$$

$$\vec{J} = \vec{J}(x,y,t), \vec{K} = \vec{K}(x,y,t) \tag{6.18}$$

With the information above and an additional assumption that the media is *isotropic instantaneously local reacting*. It is well known that under such configuration, the EM field can be decoupled into *parallel polarization* and *perpendicular polarization* part.

6.3.1 Parallel polarization

In the parallel polarization, the components $E_x(t), E_y(t), J_x(t), J_y(t), H_z(t), K_z(t)$ are coupled. The time domain Maxwell equations in the partial differential equation form are:

$$\partial_y H_z(t) = J_x(t) \tag{6.19}$$

$$-\partial_x H_z(t) = J_y(t) \tag{6.20}$$

$$\partial_x E_y(t) - \partial_y E_x(t) = K_z(t) \tag{6.21}$$

$$J_x(t) = J_x^{imp}(t) + \sigma_{xx}^e E_x(t) + \varepsilon_{xx}\partial_t E_x(t) \tag{6.22}$$

$$J_y(t) = J_y^{imp}(t) + \sigma_{yy}^e E_y(t) + \varepsilon_{yy}\partial_t E_y(t) \tag{6.23}$$

$$K_z(t) = K_z^{imp}(t) + \sigma_{zz}^m H_z(t) + \mu_{zz}\partial_t H_z(t) \tag{6.24}$$

Since the electric field is parallel to (x, y) field. This electromagnetic field is called *parallel polarization*. Since the magnetic field is pointing to the z direction. This electromagnetic field is also called *H-polarization*.

6.3.2 Perpendicular polarization

In the perpendicular polarization, the components $H_x(t), H_y(t), K_x(t), K_y(t), E_z(t), J_z(t)$ are coupled. The time domain Maxwell equations in the partial differential equation form are:

$$-\partial_y E_z(t) = K_x(t) \tag{6.25}$$

$$\partial_x E_z(t) = K_y(t) \tag{6.26}$$

$$\partial_x H_y(t) - \partial_y H_x(t) = J_z(t) \tag{6.27}$$

$$K_x(t) = K_x^{imp}(t) + \sigma_{xx}^m H_x(t) + \mu_{xx}\partial_t H_x(t) \tag{6.28}$$

$$K_y(t) = K_y^{imp}(t) + \sigma_{yy}^m H_y(t) + \mu_{yy}\partial_t H_y(t) \tag{6.29}$$

$$J_z(t) = J_z^{imp}(t) + \sigma_{zz}^e H_z(t) + \varepsilon_{zz}\partial_t E_z(t) \tag{6.30}$$

Since the electric field is pointing to z direction, perpendicular to (x, y) plane. This electromagnetic field is called *perpendicular polarization*. This electromagnetic field is also referred to as *E-polarization*.

6.4 Geometrical Discretization

In finite element package, geometrical discretization is done by mesh generator. With a non-uniform mesh, we have a good control on the coarseness of the mesh. In particular, the mesh should be refined near the interfaces, so that the numerical error can be minimized; meanwhile, the mesh should be coarser where the change in configuration is smooth. It also has been shown that equilateral triangles are the best shapes of finite elements in terms of numerical dispersion. Any triangle that is close to being equilateral introduces small numerical dispersion.

In practice, a non-uniform mesh is normally needed. But in our simple 2D configuration, a uniform mesh is satisfactory. Our system builder and solver are implemented completely separated from the mesh generator. So, if needed, the mesh generation part can be replaced by any advanced one as long as it produces the mesh file in the format that we specify.

To produce a simple uniform mesh, we shall first divide the domain into squares and then divide each of this square into two triangles, see figure 6.1.

To identify each geometrical element, including vertexes, edges and triangles, in the mesh, it is sufficient to uniquely number each vertex. And then edges will be uniquely identified with two vertexes delimiting it. Similarly, a triangle will be identifies with three vertexes delimiting it. We will number the vertexes simply as in figure 6.2.

However, due to practical reason, when generating a mesh, the mesh generator does not have a complete view of the whole domain. So it makes sense to define the numbering schema locally (equivalent to

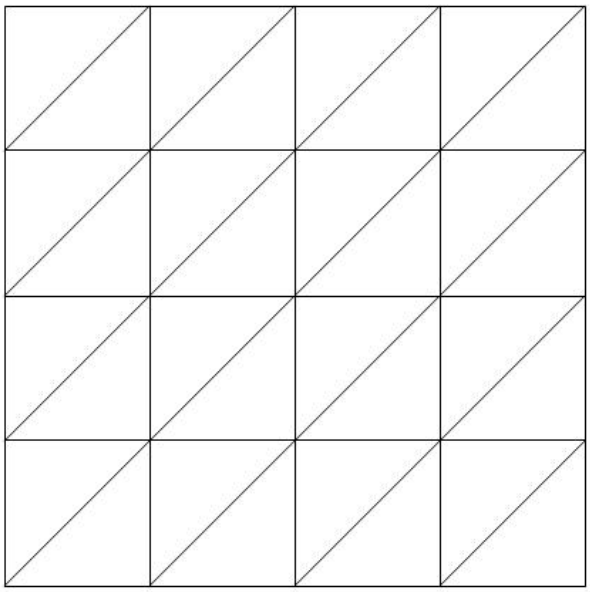

Figure 6.1: Uniform triangular mesh

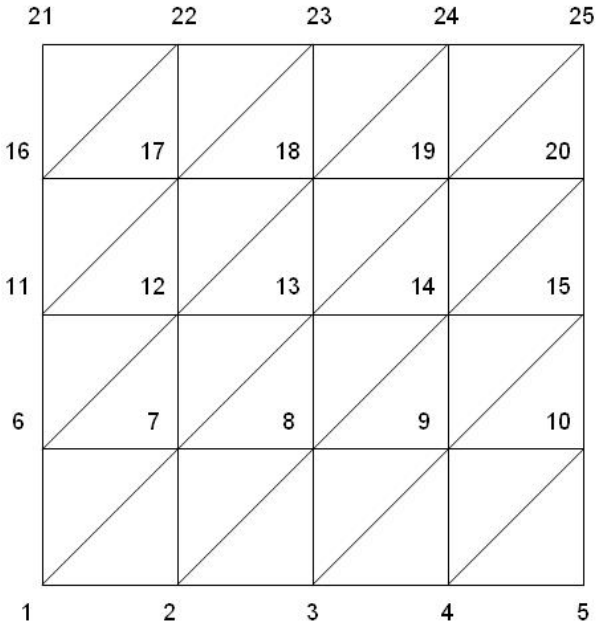

Figure 6.2: Uniform triangular mesh with numbering

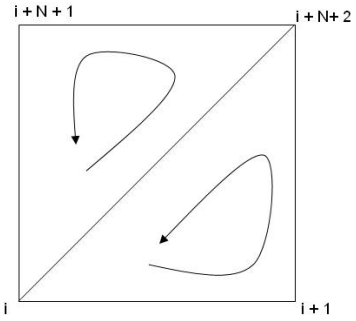

Figure 6.3: Uniform triangular mesh with local numbering

its global counterpart). Assuming the mesh has N squares in x direction and M squares in y direction, the numbering schema illustrated in figure 6.2 can be expressed locally as in figure 6.3, with the very left bottom vertex numbered as 1.

As we mentioned before, a finite element is identified with three vertexes delimiting it. It is advantageous to order the vertexes in such a way that the face vector of the all the simplexes point to the same direction (either positive z or negative z, here we choose positive z.). This implies that in figure 6.3, the two elements should be identified as $(i, i+N+2, i+N+1)$ and $(i, i+1, i+N+2)$.

It is also advantageous to keep the geometrical information and material information separately. We propose implementing material information as functions taking space coordinates and time coordinate as parameters and returning material parameters and source information; that is:

$$\vec{\sigma}^m = \vec{\sigma}^m(x,y,t), \vec{\sigma}^e = \vec{\sigma}^e(x,y,t), \vec{\varepsilon} = \vec{\varepsilon}(x,y,t), \vec{\mu} = \vec{\mu}(x,y,t) \tag{6.31}$$

$$\vec{J}^{imp} = \vec{J}^{imp}(x,y,t), \vec{K}^{imp} = \vec{K}^{imp}(x,y,t) \tag{6.32}$$

6.5 Local Field Quantities discretization for perpendicular polarization

Across the spatial discontinuous interface denoted by I, the tangential components of the electromagnetic field strengths are continuous; the normal components are free to jump across the interface. With this discontinuity, a straight-forward application of the Cartesian expansion functions would lead to large numerical error in computing field quantities or excessive mesh refinement. It is better to take this into account when discretizing the field quantities than dealing with it after. Among other things, one key point in the discretization technique is to expend (components of) the field quantities, which are known to be continuous, with straight-forward Cartesian expansion functions [21], and the discontinuous ones are expended with the edge and face expansion function defined in [20]. From this point on, we work on triangular finite elements, of which the vertexes are identified globally and uniquely as i, j, k, see figure 6.4.

However, in figure 6.4, we only look into the *perpendicular polarization* problem, where $H_x, H_y, K_x, K_y, E_z, J_z$ are coupled. Similar analysis can be done for the *parallel polarization* problem; for details, please refer to [20].

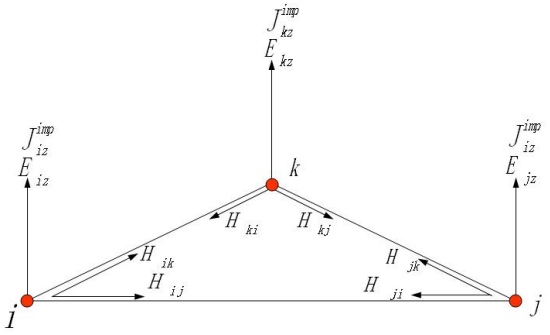

Figure 6.4: The triangular finite element on xy plane (a)

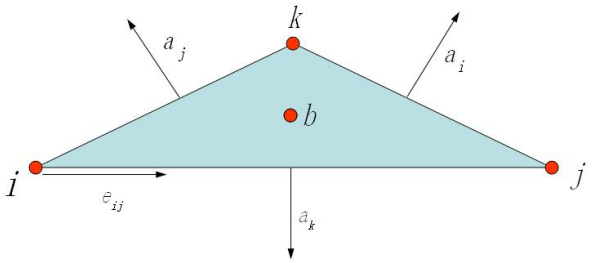

Figure 6.5: The geometrical quantities in finite element

6.5.1 Geometrical quantities

Before introducing the edge and face expansion functions, a few geometrical quantities need to be defined, see figure 6.5.

$r_i =$ the spatial coordinate of the vertex i, it is consisted of a 3 dimensional vector of x, y, z coordinates,
$l_{ij} = |r_j - r_i| =$ the length of the edge delimited by vertexes i and j,
$l_i = l_{jk} =$ the length of the edge (j, k), which does not contain i, in the triangle $\Delta(r_i, r_j, r_k)$,
$e_{ij} = \frac{r_j - r_i}{|r_j - r_i|} =$ the unit vector pointing from vertex i to j,
$\Delta(r_i, r_j, r_k) =$ the triangle delimited by three vertexes with coordinates r_i, r_j, r_k,
r_b in $\Delta(r_i, r_j, r_k) = \frac{r_i + r_j + r_k}{3} =$ the barycenter of $\Delta(r_i, r_j, r_k)$,
$a_{ij} = i_z \times e_{ij} =$ the normal unit vector perpendicular to the edge delimited by vertex i and j,
$a_k = a_{ij} =$ the normal unit vector of the edge that does not contain vertex k in the triangle $\Delta(r_i, r_j, r_k)$,
$|\Delta(r_i, r_j, r_k)| =$ the value of the surface area within the triangle $\Delta(i, j, k)$,
$\phi_i(r) = \frac{|\Delta(r, r_j, r_k)|}{|\Delta(r_i, r_j, r_k)|}, r \in \Delta(r_i, r_j, r_k)$

6.5.2 Magnetic Field Strength Discretization

With the assumption that there is no surface electric currents on the interface, the tangential component of $\vec{H}$ is continuous across the interface. We shall only use the well defined continuous component ($\vec{H}_{lh}$ where $l, h \in (i, j, k), l \neq h$, see figure 6.4) on vertexes to construct a linear interpolation over any triangle $\Delta(r_i, r_j, r_k)$.

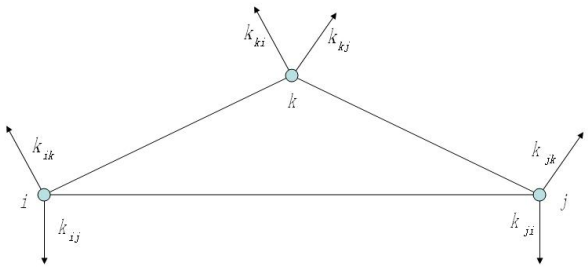

Figure 6.6: The triangular finite element on xy plane (b)

First with the continuous component of magnetic field strength on edges. The magnetic field strengths on the vertexes are well defined as:

$$\vec{H}_i = \vec{H}_{ij}\frac{1}{e_{ij}a_j}a_j + \vec{H}_{ik}\frac{1}{e_{ik}a_k}a_k = \sum_{l\in(i,j,k),l\neq i}\vec{H}_{il}\frac{1}{e_{il}a_l}a_l \tag{6.33}$$

With the magnetic field strengths defined on each vertexes, the cartesian expansion functions can be used to construct the linear interpolation $[\vec{H}]_r$ of $\vec{H}_r, r \in \Delta(r_i, r_j, r_k)$, as follows:

$$[\vec{H}]_r = \sum_{l\in(i,j,k)}\vec{H}_l\phi_l(r) = \sum_{l\in(i,j,k)}\sum_{h\in(i,j,k),h\neq l}\vec{H}_{lh}\frac{1}{e_{lh}a_h}a_h\phi_l(r) \tag{6.34}$$

6.5.3 Volume density of Magnetic volume density Discretization

With the assumption that there is no surface electric currents on the interface, the normal component of $\vec{K}$ is continuous across the interface. We shall only use the well defined continuous component ($\vec{K}_{lh}$ where $l, h \in (i, j, k), l\neq h$, see figure 6.6) on vertexes to construct a linear interpolation over any triangle $\Delta(r_i, r_j, r_k)$.

First with the continuous component of magnetic field strength on edges. The magnetic field strengths on the vertexes are well defined as:

$$\vec{K}_i = \vec{K}_{ij}\frac{1}{e_{ij}a_j}e_{ij} + \vec{K}_{ik}\frac{1}{e_{ik}a_k}e_{ik} = \sum_{l\in(i,j,k),l\neq i}\vec{K}_{il}\frac{1}{e_{il}a_l}e_{il} \tag{6.35}$$

With the magnetic field strengths defined on each vertexes, the cartesian expansion functions can be used to construct the linear interpolation $[\vec{K}]_r$ of $\vec{K}_r, r \in \Delta(r_i, r_j, r_k)$, as follows:

$$[\vec{K}]_r = \sum_{l\in(i,j,k)}\vec{K}_l\phi_l(r) = \sum_{l\in(i,j,k)}\sum_{h\in(i,j,k),h\neq l}\vec{K}_{lh}\frac{1}{e_{lh}a_h}e_{lh}\phi_l(r) \tag{6.36}$$

However, it will become clear that: when this local interpolation is expanded to global one, not only the value of K has to be continuous, but also the direction has to be continuous. Applying this local expansion directly would result in discontinuity of direction in the two finite elements which share a same edge. The trick is to take the sign of K_{ij} defined on the common edge $\varepsilon(i, j)$ to be different in the two adjacent triangles. Then both the value of K_{ij} and the direction are uniquely defined.

Note that, there is no such problem for other interpolated quantities in perpendicular polarization case, because the discretization technique does not incur ambiguity in their directions.

6.5.4 Electric Field Strength Discretization

Due to the fact the in perpendicular polarization, the electric field points to the z direction, which is always tangential to the interface. Hence, E_r points to z direction and always continuous. Therefore, the electric field strengths on each vertex are well defined. A cartesian expansion function can be used to construct the linear interpolation $[E]_r$ of E_r in $\Delta(r_i, r_j, r_k)$.

$$\vec{E}_r = \sum_{l \in (i,j,k)} \vec{E}_l \phi_l(r) \tag{6.37}$$

E_r points to z direction, Hence:

$$[\vec{E}]_r = \sum_{l \in (i,j,k)} \vec{E}_l i_z \phi_l(r) \tag{6.38}$$

6.5.5 Volume Density of Electric Current Discretization

Due to the fact the in perpendicular polarization, the electric current points to the z direction, which is always tangential to the interface. Hence, E_r points to z direction and always continuous. Therefore, the electric current densities on each vertex are well defined. A cartesian expansion function can be used to construct the linear interpolation $[J]_r$ of J_r in $\Delta(r_i, r_j, r_k)$.

$$[\vec{J}]_r = \sum_{l \in (i,j,k)} \vec{J}_l \phi_l(r) \tag{6.39}$$

J_r points to z direction, Hence:

$$[\vec{J}]_r = \sum_{l \in (i,j,k)} \vec{J}_l i_z \phi_l(r) \tag{6.40}$$

6.5.6 Material Parameter Discretization

The cartesian expansion function can be used directly to construct the linear interpolation $[\bar{Q}]_r$ of $\bar{Q}_r$ in $\Delta(r_i, r_j, r_k)$.

$$[\bar{Q}]_r = \sum_{l \in (i,j,k)} \bar{Q}_l \phi_l(r) \tag{6.41}$$

Q represents one of these material parameters.

6.6 Temporal discretization

Assuming that all field strengths vanish before the time point 0, we simply define the $\partial_t Q(t)$ with the backward law as follows:

$$[\partial_t Q(t)] = \frac{Q(t) - Q(t - \Delta t)}{\Delta t} \tag{6.42}$$

The time step Δt is taken to be:

$$\Delta t \leq \frac{\lambda}{2c_0} \tag{6.43}$$

where, λ is the maximal length of the cells, c_0 is the speed of light in vacuum, see [28]. The size of the finite element is controlled by the mesh generator with respect to frequency of the sources.

6.7 System formulation

With all these interpolated field quantities in hand, the continuous field quantities in the integrated Maxwell equations can be replaced by these linearly interpolated quantities. Since the interpolated quantities are linear, the integral equation can be simplified into simple operations. For a detailed derivation, refer to [20]. Note that in [20], the discretized Maxwell equations work in frequency domain, here we work in time-domain. The derivation does not differ from the frequency domain ones, except s operations are replaced by ∂_t, which is further replaced by its temporally discretized counterpart.

There are two ways of applying the integrated Maxwell equations, namely, Volume Integrated Field Equations and Surface Integrated Field Equations. We will go though these two methods in case of *perpendicular polarization* . And we assume that the media is *isotropic instantaneously reacting*.

6.7.1 Volume Integrated Field Equations

Detailed derivation is given in Jorna's thesis, here we give a survey on these equations. The Maxwell equations at time t applied in Volume Integrated Field Equations require:

- Applying the equation 6.27 in its integrated form gives:

$$\frac{1}{2}l_i[H_{jk}(t) - H_{kj}(t)] + \frac{1}{2}l_j[H_{ki}(t) - H_{ik}(t)] + \frac{1}{2}l_k[H_{ij}(t) - H_{ji}(t)]$$

$$-\frac{1}{A}\{[\sigma^e_{izz} + \varepsilon_{izz}\partial_t]E_{iz}(t) + [\sigma^e_{jzz} + \varepsilon_{jzz}\partial_t]E_{jz}(t) + [\sigma^e_{kzz} + \varepsilon_{kzz}\partial_t]E_{kz}(t)\}$$

$$= \frac{1}{A}[J^{imp}_{iz}(t) + J^{imp}_{jz}(t) + J^{imp}_{kz}(t)] \tag{6.44}$$

  further with temporally discretization, it means:

$$\frac{1}{2}l_i[H_{jk}(t) - H_{kj}(t)] + \frac{1}{2}l_j[H_{ki}(t) - H_{ik}(t)] + \frac{1}{2}l_k[H_{ij}(t) - H_{ji}(t)]$$

$$-\frac{1}{A}[(\sigma^e_{izz} + \frac{\varepsilon_{izz}}{\Delta t})E_{iz}(t) + (\sigma^e_{jzz} + \frac{\varepsilon_{jzz}}{\Delta t})E_{jz}(t) + (\sigma^e_{kzz} + \frac{\varepsilon_{kzz}}{\Delta t})E_{kz}(t)]$$

$$= \frac{1}{A}[J^{imp}_{iz}(t) + J^{imp}_{jz}(t) + J^{imp}_{kz}(t) - \frac{\varepsilon_{izz}}{\Delta t}E_{iz}(t-\Delta t) - \frac{\varepsilon_{jzz}}{\Delta t}E_{jz}(t-\Delta t) - \frac{\varepsilon_{kzz}}{\Delta t}E_{kz}(t-\Delta t)] \tag{6.45}$$

- Applying the equation 6.25 and 6.26 in its volume integrated form gives:

$$E_{iz}(t) - E_{kz}(t) + \frac{1}{3}[(\pm)l_jK_{ij}(t) - (\pm)l_iK_{ji}(t) + (\pm)l_jK_{kj}(t) - (\pm)l_kK_{jk}(t)] = 0 \tag{6.46}$$

$$E_{iz}(t) - E_{jz}(t) + \frac{1}{3}[(\pm)l_iK_{ki}(t) - (\pm)l_kK_{ik}(t) + (\pm)l_jK_{kj}(t) - (\pm)l_kK_{jk}(t)] = 0 \tag{6.47}$$

- The compatibility relation requires that:

$$l_i[(\pm)K_{ji}(t) + (\pm)K_{ki}(t)] + l_j[(\pm)K_{ij}(t) + (\pm)K_{kj}(t)] + l_k[(\pm)K_{i,k}(t) + (\pm)K_{j,k}(t)] = 0 \tag{6.48}$$

The $(\pm)$ signs in these equations are chosen to uniquely determine the direction of K_{xy}. If the current element is the first element referring to K_{xy}, the plus sign$(+)$ is chosen, otherwise the minus sign$(-)$.

6.7.2 Surface Integrated Field Equations

Detailed derivation is given in Jorna's thesis, here we give a survey on these equations. The Maxwell equations at time t applied in Surface Integrated Field Equations require:

- Applying the equation 6.27 in its integrated form gives:

$$\frac{1}{2}l_i[H_{jk}(t) - H_{kj}(t)] + \frac{1}{2}l_j[H_{ki}(t) - H_{ik}(t)] + \frac{1}{2}l_k[H_{ij}(t) - H_{ji}(t)]$$

$$-\frac{1}{A}[(\sigma^e_{izz} + \frac{\varepsilon_{izz}}{\Delta t})E_{iz}(t) + (\sigma^e_{jzz} + \frac{\varepsilon_{jzz}}{\Delta t})E_{jz}(t) + (\sigma^e_{kzz} + \frac{\varepsilon_{kzz}}{\Delta t})E_{kz}(t)]$$

$$= \frac{1}{A}[J^{imp}_{iz}(t)+J^{imp}_{jz}(t)+J^{imp}_{kz}(t)-\frac{\varepsilon_{izz}}{\Delta t}E_{iz}(t-\Delta t)-\frac{\varepsilon_{jzz}}{\Delta t}E_{jz}(t-\Delta t)-\frac{\varepsilon_{kzz}}{\Delta t}E_{kz}(t-\Delta t)] \quad (6.49)$$

- Applying the equation 6.25 and 6.26 in its Surface integrated form gives:

$$E_k(t) - E_j(t) = -\frac{1}{2}l_i[(\pm)K_{ji}(t) + (\pm)K_{ki}(t)] \quad (6.50)$$

$$E_i(t) - E_k(t) = -\frac{1}{2}l_j[(\pm)K_{ij}(t) + (\pm)K_{kj}(t)] \quad (6.51)$$

$$E_j(t) - E_i(t) = -\frac{1}{2}l_k[(\pm)K_{ik}(t) + (\pm)K_{jk}(t)] \quad (6.52)$$

The $(\pm)$ signs in these equations are chosen to uniquely determine the direction of K_{xy}. If the current element is the first element referring to K_{xy}, the plus sign$(+)$ is chosen, otherwise the minus sign$(-)$.

6.7.3 Constitutive Relations

To determine the electromagnetic field strength, the constitutive relations described via equation 6.28 and 6.29 are needed. Here in the Integrated Field Equation method, they are defined as:

$$(\pm)K_{ij}(t)\frac{e_{ij}i_a}{e_{ij}a_j} + K_{ik}(t)\frac{e_{ik}i_a}{e_{ik}a_k} - (\sigma^m_{iaa} + \mu_{iaa}\partial_t)[H_{ij}(t)\frac{a_j i_a}{e_{ij}a_j} + H_{ik}(t)\frac{a_k i_a}{e_{ik}a_k}] = \vec{K}^{imp}_i(t)i_a$$

$$i{\neq}j{\neq}k, a = x, y \quad (6.53)$$

If the magnetic constitutive parameters are isotropic with respect to x and y direction; the above equations can be simplified as:

$$(\pm)K_{ij}(t) - (\sigma^m_i + \mu_i\partial_t)[H_{ij}(t)\frac{1}{e_{ij}a_j} + H_{ik}(t)\frac{a_k a_j}{e_{ik}a_k}] = \vec{K}^{imp}_i(t)a_j$$

$$i{\neq}j{\neq}k \quad (6.54)$$

With temporally discretization the above equation will be discretized in time domain as:

$$(\pm)K_{ij}(t)-(\sigma^m_i+\frac{\mu_i}{\Delta t})[H_{ij}(t)\frac{1}{e_{ij}a_j}+H_{ik}(t)\frac{a_k a_j}{e_{ik}a_k}] = \vec{K}^{imp}_i(t)a_j-\frac{\mu_i}{\Delta t}[H_{ij}(t-\Delta t)\frac{1}{e_{ij}a_j}+H_{ik}(t-\Delta t)\frac{a_k a_j}{e_{ik}a_k}]$$

$$i{\neq}j{\neq}k \quad (6.55)$$

The $(\pm)$ signs in these equations are chosen to uniquely determine the direction of K_{xy}. If the current element is the first element referring to K_{xy}, the plus sign$(+)$ is chosen, otherwise the minus sign$(-)$.

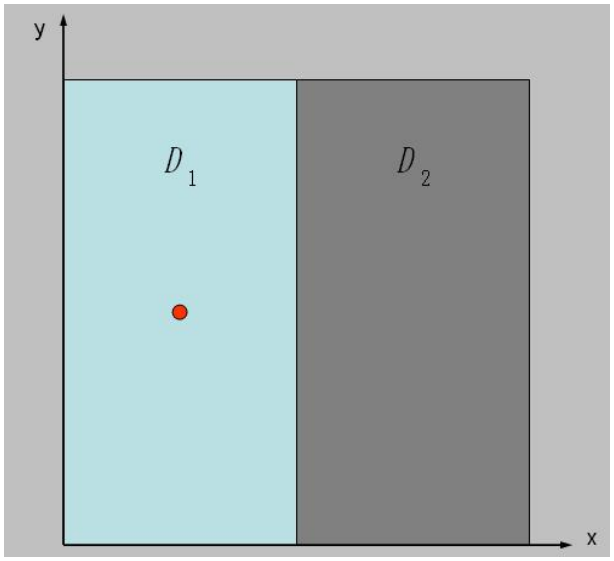

Figure 6.7: Configuration consisting of two homogeneous subdomains with different medium properties.

6.7.4 System Equations

With appropriate boundary conditions and initial field quantities at time point 0, the electromagnetic field strengths in case of *perpendicular polarization* can be solved with Volume Integrated Equations 6.44, 6.45, 6.46, 6.48 and constitutive relations 6.54, or with Surface Integrated Equations 6.49, 6.50, 6.51, 6.52 and constitutive relations 6.54. If the media is further isotropic, the constitutive relations can be simplified as 6.55. Although the VIFEs and SIFEs actually describe the same equations, they will result in very different system matrices. This will has strong influence on the system solver.

6.8 Test configuration

In order to understand the behavior of the EM modeling method as well as the solution method, a simple test case, of which the analytic solution is available, is needed. In this thesis, we restrict ourselves to 2D configuration in which the perpendicular polarization and parallel polarization are decoupled.

6.8.1 Computation Domain

The configuration used for testing is a square domain $D = \{0 \leq x \leq 1, 0 \leq y \leq 1\}$, see figure 6.7, consisting of two homogenous subdomains $D_i, i = 1, 2$ with different medium properties $\sigma_1^e, \sigma_1^m, \varepsilon_1, \mu_1$ and $\sigma_2^e, \sigma_2^m, \varepsilon_2, \mu_2$. There is a tiny line current running in z direction. The current density is described with the function $J = J(x, y, t)$; where t specifies the time coordinate. Without losing generality, we assume $J(x, y, t) = 0$ for $t < 0$. Due to the fact that we are working on space-time domain, the configuration does not have to be time-harmonic as in [20].

Since, in our configuration, we only have current density in z direction, that is we only have *perpendicular polarization* . Given enough information on materials and boundary, the electromagnetic field can be determined using the equations 6.25 - 6.30.

6.8.2 Boundary Conditions

In the subdomains where the constitutive parameters are continuous with respect to spatial coordinate, the electromagnetic strengths are also continuously differentiable functions of position. Across certain boundary surface in the configuration, certain component of the electromagnetic strength exhibit discontinuity. These interface problems inside the computation domain have been taken care of by the specify discretization which only uses the well defined continuous components. A outer boundary condition is needed to truncate the computation domain, and uniquely define the electromagnetic field strengths inside the computation domain ,with the help of the Maxwell equations. Many types of boundary conditions can be applied, as long as they satisfy a uniqueness condition. Here for this simple test configuration we use *prescribed tangential electric field* boundary condition to truncate the domain $D_1 \cup D_2$. More specifically, we choose tangential electric field on the outer boundary as 0. Since, in case of *perpendicular polarization*, the electric field points to z direction which is always tangential to the boundary surface, the electric field vanishes in the outer domain ($\bar{D_1} \cap \bar{D_2}$).

6.9 System matrix

With the test configuration specified in section 6.8, we apply the Maxwell equations in the Surface Integrated Field Equations ways. Then, we walk though each triangular finite element and build the system equations. As for the numbering of unknowns, we first number the unknowns on vertexes and then the unknowns on edges in each finite element with the scanning order. This numbering schema is however open doubt. But as far as for 2D problem, this gives a well arranged matrix (data entries are concentrated on the diagonal). We are also experimenting with new numbering schemes such that the pattern of system matrices could be more favorable. Note that the same unknown should not be numbered twice. The known component of the electromagnetic field strength (ie, the tangential electric field strength on the outer boundary) will not be numbered as unknowns.

Now let us have a look at how large the system matrix will be. Let us suppose we have N cells on the x direction and M cells on the y direction. All the cells are further divided into triangles. As we are applying the SIFE equations and using the *prescribed tangential electric field* condition to formulate the system in case of *perpendicular polarization*. We shall have 10 equations per each finite element. We shall have 1 unknown on each vertex (E_z), four unknowns on each edge $H_{ij}, H_{ji}, K_{ij}, K_{ji}$. And on the outer boundary, the E_zs are prescribed, therefore, these E_zs on the outer boundary should not be counted as unknowns.

Hence, the system matrix is of size $X \times Y$, where

$$X = 10 \times \text{(number of the finite elements)} = 20NM$$

$$Y = \text{(number of vertexes)} + 4\text{(number of edges)} - \text{(number of vertexes on the outer boundary)}$$

$$= (N+1)(M+1) + 4[N(M+1) + M(N+1) + NM] - 2(N+M) = 13NM + 3N + 3M + 1$$

By simply looking into the formulas for X and Y, one can easily find out that $X > Y$. This indicates a overdetermined system. with the methodology we described above, the system matrix of any mesh can be written down. We shall display the system matrices for 3×3 grid, 5×5 grid, 10×10 grid, and 100×100 grid in figures 6.8, 6.9, 6.10, 6.11 . These give a good overview on the pattern of the system matrices.

It then becomes quite obvious that the system matrix in case of *perpendicular polarization* is a *sparse double band overdetermined* matrix.

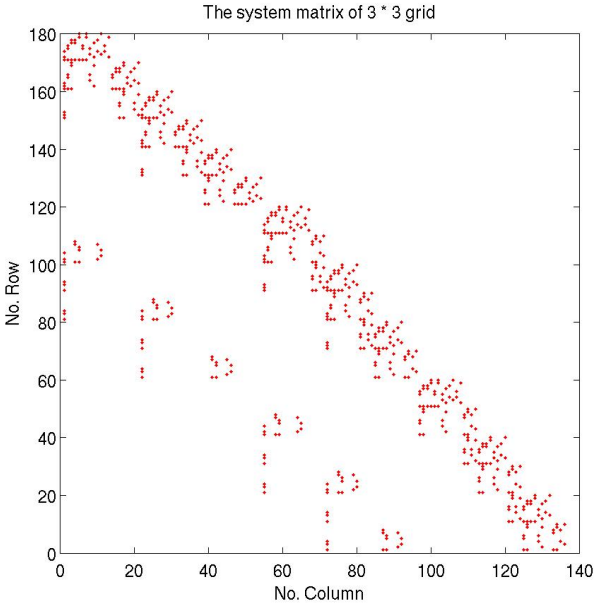

Figure 6.8: The system matrix of 3×3 grid

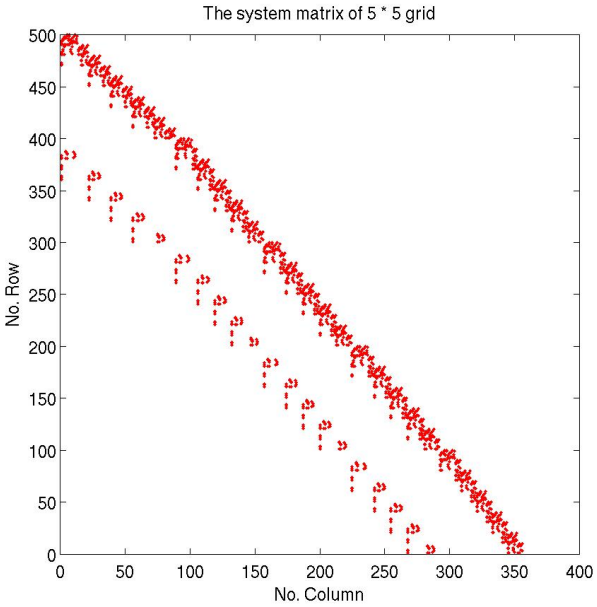

Figure 6.9: The system matrix of 5×5 grid

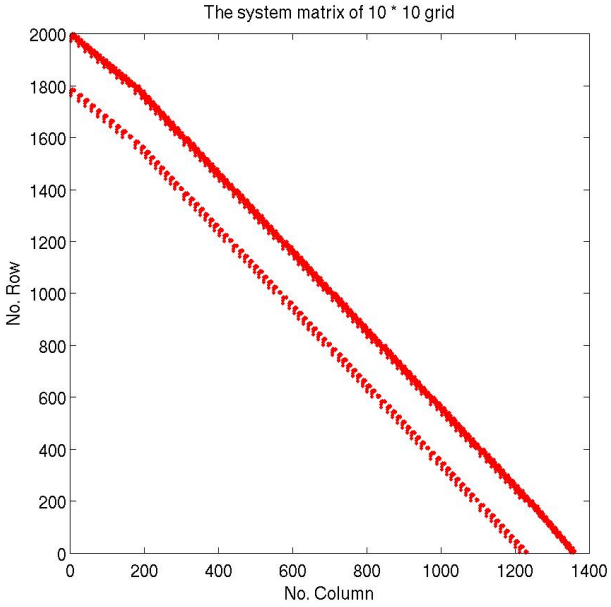

Figure 6.10: The system matrix of 10×10 grid

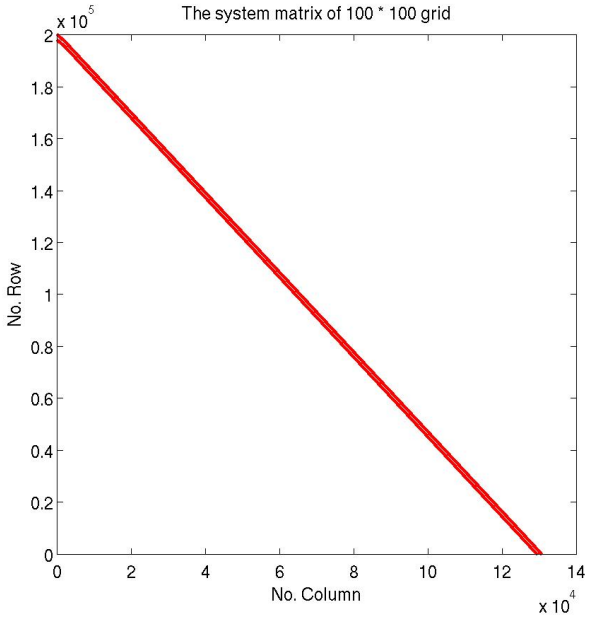

Figure 6.11: The system matrix of 100×100 grid

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we presented the The HSS theory, pioneered by Chandrasekaran and Gu [6]. A number of its fast algorithms have been presented as modifications on dataflow diagrams. These dataflow diagrams serve as a useful tool to understand and analyze these complicated HSS algorithms. We also discussed the construction algorithms of the conventional HSS representation. Although no practical construction algorithm is given in this thesis, we assume that we can get the HSS representation efficiently from the practical system. We also developed one variant of the HSS representation, we believe it brings more flexibility. A number of algorithms for this extended version of HSS representation have been developed, but out of the scope of this thesis.

A number of matrix operations based on HSS representation, which are either collected from the literature or new, have been introduced and implemented. Their behavior have been studied. Numerical experiments suggest that all these algorithms have linear complexity in computation time and more importantly in memory usage, given the system matrix is *Hierarchically Semi-separable*. We also studied and the relation among the HSS representation and SSS representation or equivalently the Time-varying representation. Conversion formulas among them have been given.

The limitation of these direct solution method has been studied. A general strategy to combine the HSS representation with iterative solution algorithms has been given. With this strategy, any iterative algorithm can be easily combined with the HSS representations. A number of iterative solution algorithms based on HSS representations have been implemented and tested. All suggest that: when the off-diagonal blocks of the system matrix are not so smooth, the iterative algorithms based HSS representations exceed its direct counterpart in CPU time memory usage. A number of *preconditioners* based on HSS representation have been proposed to improve the convergence of the iterative methods. Numerical experiments have proven its usefulness.

As for the practical part, we implement the Integrated Field Equations Method on triangular finite element mesh to simulate the electromagnetic field in case of perpendicular polarization. A simple 2D case is tested and the system matrix has been generated. Yet we have not been successful in combining the solution method with this practical problem.

7.2 Future Work

In the last section of this thesis, we recommend some future research direction.

First of all, although efficient algorithms can be given based on the HSS representation, getting this HSS representation from the system is not trivial. A good construction algorithm should be developed to utilize the geometric information of the system. For instance, efficient construction algorithm can be derived for 1D FMM problem. However, the structure of HSS becomes too restricted for more complicated problems. Extension should be made on the original HSS representation to improve its applicability (The same way as we extend SSS to get HSS).

Secondly, due to the local elimination strategy of the HSS direct solver, it is not capable of handling Moore-Penrose solution. A theoretical solution has been given by P.Dewilde [7], yet, a practical implementation has not been developed.

Thirdly, for the practical part, as we planed, we will extend the 2D version of the field wave solver to its 3D version. Trapezium-Law should be used for temporal discretization. However, this would result in even more equations and more unknowns. How to solve such system efficiently is the crucial problem. Unfortunately, the HSS algorithms turned to be not suitable for the system matrix it results.

The last recommendation goes to the model reduction methods for HSS representation. The method we are using now is a heuristic one. Although it reduces the HSS complexit, it does not provide the best approximation. More theories can be derived under the HSS framework.

Bibliography

[1] C. Ashcraft and J. W. H. Liu. A partition improvement algorithm for generalized nested dissection. *Tech. Rep. BCSTECH-94-020, Boeing Computer Services, Seattle, WA*, 1994.

[2] Owe Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[3] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. *in Proc.Parallel Processing for Scientific Computing, SIAM, Philadelphia*, pages 445–452, 1993.

[4] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals. A fast solver for hss representations via sparse matrices. In *Technical Report*. Delft University of Technology, August 2005.

[5] S. Chandrasekaran, M Gu, and T. Pals. A fast and stable solver for smooth recursively semi-separable systems. In *SIAM Annual Conference, San Diego and SIAM Conference of Linear Algebra in Controls, Signals and Systems, Boston*, 2001.

[6] S. Chandrasekaran, M. Gu, and T. Pals. Fast and stable algorithms for hierarchically semi-separable representations. In *Technical Report*. University of California at Santa Barbara, April 2004.

[7] Patrick Dewilde and Shiv Chandrasekaran. A hierarchical semi-separable moore-penrose equation solver. Nov 2005.

[8] P. Dewilde, K. Diepold, and W. Bamberger. A semi-separable approach to a tridiagonal hierarchy of matrices with application to image flow analysis. In *Proceedings MTNS*, 2004.

[9] P. Dewilde and A.-J. van der Veen. *Time-varying Systems and Computations*. Kluwer, 1998.

[10] P. Dewilde and A.-J. van der Veen. Inner-outer factorization and the inversion of locally finite systems of equations. *Linear Algebra and its Applications*, 313:53–100, 2000.

[11] Y. Eidelman and I. Gohberg. On a new class of structured matrices. *Notes distributed at the 1999 AMS-IMS-SIAM Summer Research Conference*, Structured Matrices in Operator Theory, Numerical Analysis, Control, Signal and Image Processing, 1999.

[12] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal*, 10:345–363, 1973.

[13] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Rev.*, 31:1–19, 1989.

[14] I. Gohberg, T. Kailath, and I. Koltracht. Linear complexity algorithms for semiseparable matrices. *Integral Equations and Operator Theory*, 8:780–804, 1985.

[15] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys.*, 73:325–348, 1987.

[16] S.Chandrasekaran P.Dewilde Ming Gu and Kaviyesh Doshi. Personal notes on dataflow diagrams. September 2005.

[17] S.Chandrasekaran Z.Sheng P.Dewilde Ming Gu and Kaviyesh Doshi. Hierarchically semi-separable rrepresentation and dataflow diagrams. Nov 2005.

[18] W. Hackbusch. A sparse arithmetic based on $\mathcal{H}$-matrices. part i: Introduction to $\mathcal{H}$-matrices. *Computing*, 64:21–47, 2000.

[19] BRUCE HENDRICKSON and EDWARD ROTHBERG. Improving the run time and quality of nested dissection ordering. *SIAM J. SCI. COMPUT.*, 20:468–489, 1998.

[20] Pieter Jorna. Integrated field equations methods for the computation of electromagnetic fields in strongly inhomogeneous media. *PHD thesis, TUDelft*, Feb 2005.

[21] Ioan E. Lager and Gerrit Mur. Generalized cartesian finite elements. *IEEE TRANSACTIONS ON MAGNETICS*, 34(4):2220–2227, july 1998.

[22] W.Hackbusch leipzig. A sparse matrix arithmetic based on h-matrices. part 1: Introduction to h-matrices. *Computing*, December 1998.

[23] William Lyons. Fast algorithms with applications to pdes. *PHD thesis*, June 2005.
[24] S.Chandrasekaran M.Gu and T.Pals. A fast ulv decomposition solver for hierachically semiseparable representations. 2004.
[25] S.Chandrasekaran M.Gu and W.Lyons. A fast and stable adaptive solver for hierachically semi-separable representations. April 2004.
[26] T. Pals. *Multipole for Scattering Computations: Spectral Discretization, Stabilization, Fast Solvers*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, 2004.
[27] R.Barrett, M. Berry, T.F.Chan, J.Demmel, J.Donato, J.Dongarra, V.Eijhout, R.Pozo, C.Romine, and H.Van der Vorst. *Templates for the solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphina, PA, 1994.
[28] Remis Rob. Note on fdtd method. *Course note, TUDelft*, 2006.
[29] V. Rokhlin. Applications of volume integrals to the solution of pde's. *J. Comp. Phys.*, 86:414–439, 1990.
[30] W.Lyons S.Chandrasekaran and M.Gu. Fast lu decomposition for operators with hierachically semiseparable structure. April 2005.
[31] Jonathan Richard Shewchuk. *An introduction to the Congugate Gradient Method Without the Agonizing Pain*. School of Computer Science Carnegie Mellon University, 1994.
[32] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *J. Proc. IEEE*, 55:1801–1809, April 1967.
[33] S.Chandrasekaran P.Dewilde W.Lyons T.Pals and Alle-Jan van der Veen. Fast stable solver for sequentially semi-separable linear systems of equations. Octorber 2002.
[34] A.J. van der Veen. Time-varying lossless systems and the inversion of large structured matrices. *Archiv f. Elektronik u. Übertragungstechnik*, 49(5/6):372–382, Sept. 1995.
[35] S.Chandrasekaran P.Dewilde M.Gu W.Lyons T.Pals Alle-Jan van der Veen and Jianlin Xia. A fast backward stable solver for sequentially semi-separable matrices. September 2005.
[36] V.Faber and T.Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J*, Numer Anal.21:315–339, 1984.
[37] S.Chandrasekaran P.Dewilde M.Gu W.Lyons and T.Pals. A fast solver for hss representation via sparse matrices. Auguest 2005.

Appendix A

pseudo-code of iterative methods

A.1 Preconditioned Conjugated gradient method based on HSS algorithms

```
Compute r^(0) = b − Ax^(0) for some initial guess x^(0)
for i = 1, 2, ...
   Solve Mz^(i−1) = r^(i−1)
   ρ_{i−1} = r^(i−1)T z^(i−1)
   if i = 1
      p^(1) = z^(0)
   else
      β_{i−1} = ρ_{i−1}/ρ_{i−2}
      p^(i) = z^(i−1) + β_{i−1} p^(i−1)
   endif
   q^(i) = Ap^(i)
   α_i = ρ_{i−1}/p^(i)T q^(i)
   x^(i) = x^(i−1) + α_i p^(i)
   r^(i) = r^(i−1) − α_i q^(i)
   check convergence; continue if necessary
end
```

Note here that, to combine with HSS algorithms, the matrices A and M here are represented with HSS representations. Matrix-vector multiplication is done with the fast HSS matrix-vector multiplication method; If a HSS *preconditioner* is used, inverting the *preconditioner* M is done with fast HSS direct solution method. If a Block Jacobi preconditioner is used, the inverse of the preconditioner M can be applied easily on a vector block-wise.

A.2 Preconditioned BiCG method based on HSS algorithms

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\tilde{r}^{(0)}$ (for example, $\tilde{r}^{(0)} = r^{(0)}$) for $i = 1, 2, ...$
  Solve $Mz^{(i-1)} = r^{(i-1)}$
  Solve $M^T\tilde{z}^{(i-1)} = \tilde{r}^{(i-1)}$
  $\rho_{i-1} = z^{(i-1)T}\tilde{r}^{(i-1)}$
  if $\rho_{i-1} = 0$, method fails
  if $i = 1$
    $p^{(1)} = z^{(0)}$
    $\tilde{p}^{(1)} = \tilde{z}^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$
    $\tilde{p}^{(i)} = \tilde{z}^{(i-1)} + \beta_{i-1}\tilde{p}^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\tilde{q}^{(i)} = A^T\tilde{p}^{(i)}$
  $\alpha_i = \rho_{i-1}/\tilde{p}^{(i)T}q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  $\tilde{r}^{(i)} = \tilde{r}^{(i-1)} - \alpha_i \tilde{q}^{(i)}$
  check convergence; continue if necessary
end

Note here that, to combine with HSS algorithms, the matrices A and M here are represented with HSS representations. The transpose of A and M are computed with the HSS transpose algorithm described in section 4.1.3. Matrix-vector multiplication is done with the fast HSS matrix-vector multiplication method;If a HSS *preconditioner* is used, inverting the *preconditioner* M is done with fast HSS direct solution method. If a Block Jacobi preconditioner is used, the inverse of the preconditioner M can be applied easily on a vector block-wise.

A.3 Preconditioned CGS method based on HSS algorithms

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\tilde{r}^{(0)}$ (for example, $\tilde{r}^{(0)} = r^{(0)}$) for $i = 1, 2, ...$
  $\rho_{i-1} = \tilde{r}^T r^{(i-1)}$
  if $\rho_{i-1} = 0$, method fails
  if $i = 1$
    $u^{(1)} = r^{(0)}$
    $p^{(1)} = u^{(1)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $u^{(i)} = r^{(i-1)} + \beta_{i-1} q^{(i-1)}$
    $p^{(i)} = u^{(i-1)} + \beta_{i-1}(q^{(i-1)} + \beta_{i-1} p^{(i-1)})$
  endif
  solve $M\hat{p} = p^{(i)}$
  $\hat{v} = A\hat{p}$
  $\alpha_i = \rho_{i-1}/\tilde{r}^T \hat{v}$
  $q^{(i)} = u^{(i)} - \alpha_i \hat{v}$
  Solve $M\hat{u} = u^{(i)} + q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i u^{(i)}$
  $\hat{q} = A\hat{u}$
  $r^{(i)} = r^{(i-1)} - \alpha_i \hat{q}$
  check convergence; continue if necessary
end

Note here that, to combine with HSS algorithms, the matrices A and M here are represented with HSS representations. Matrix-vector multiplication is done with the fast HSS matrix-vector multiplication method;If a HSS *preconditioner* is used, inverting the *preconditioner* M is done with fast HSS direct solution method. If a Block Jacobi preconditioner is used, the inverse of the preconditioner M can be applied easily on a vector block-wise.