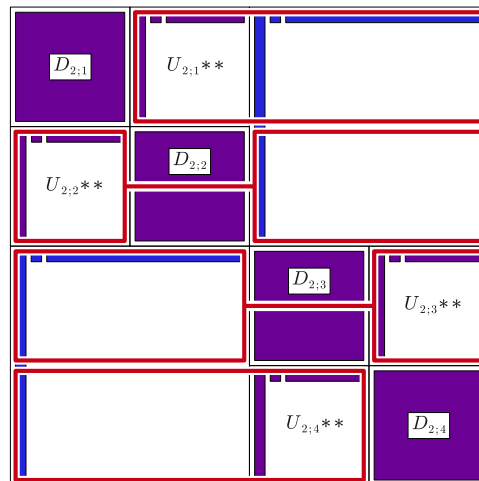


Semester Thesis

A Numerical Solver for Lyapunov
Equations Based on the Matrix Sign
Function Iteration in HSS Arithmetic

Stefan Pauli

Mai 31, 2010



Supervisor: Prof. Dr. Daniel Kressner

Abstract

A Numerical Solver for Lyapunov Equations Based on the Matrix Sign
Function Iteration in HSS Arithmetic

by

Stefan Pauli

The Hierarchically Semiseparable (HSS) representation can achieve memory efficient storage and fast computations on certain structured matrices. Matrices with no structure whatsoever can be represented in HSS form; however, it is only the case when a specific low rank structure is found that HSS can provide the above mentioned benefits. Once a matrix has been stored in HSS form, a Lyapunov equation solver ($AX + XA^T = -BB^T$) which exploits this representation can be developed.

This thesis explores the theory behind, and implementation of an iterative Lyapunov equation solver that work with HSS representations. In addition, the following operations were implemented:

- The HSS representation of a tridiagonal matrix
- An exact inverse for a triangular matrix in HSS form
- The LU decomposition of a matrix in HSS form

Contents

1	Introduction to HSS	4
1.1	Matrix and Tree Representation of HSS	4
1.2	Modular Tree Representation	6
1.3	HSS Toolbox	7
2	Algorithms for Matrices in HSS form	7
2.1	HSS Inverse of a Triangular Matrix	7
2.1.1	Introduction to Block Inverse of a Triangular Matrix .	7
2.1.2	Derivation	9
2.1.3	The Algorithm	11
2.2	HSS LU	12
2.2.1	The Algorithm	12
2.3	HSS Representation of a Tridiagonal Matrix	13
2.3.1	The Algorithm	13
3	Introduction to Sign Function Iteration	14
3.1	Iterative refinement	15
4	Sign Function Iteration in HSS Arithmetic	15
4.1	Implementation of GECLNC	15
4.1.1	Computation of the Inverse in HSS Arithmetic	15
4.1.2	Computation of c_k	16
4.1.3	Computation of the GECLNC Algorithm	16
5	Experimental Results	16
5.1	Test Setup	16
5.2	Tuning	17
5.3	Convergence	17
5.4	Speed	19
6	Summary	19
6.1	Outlook	19

1 Introduction to HSS

We follow the introduction to HSS of [1].

1.1 Matrix and Tree Representation of HSS

Here a precise mathematical description of the HSS representation is given. It is shown that there exists an equivalent HSS tree representation of the matrix in HSS representation. It will become clear that both the matrix and HSS tree representations will be necessary to understand the algorithms used with matrices in HSS representation.

We introduce the following notation for the matrix in level-zero HSS representation,

$$A = m_{0;1} \begin{pmatrix} A \end{pmatrix}^{n_{0;1}},$$

to make it obvious that A is a $m_{0;1} \times n_{0;1}$ matrix. The 0 in 0; 1 of the subscript of m and n denotes the level-zero, i.e., the matrix has not yet been partitioned. The 1 in 0; 1 will be introduced once the first URV decomposition is formed.

The HSS representation depends directly on the recursive block partitioning of the matrix A . The first partitioning will divide the matrix in four blocks. This leads to the following representation

$$A = \begin{matrix} & \begin{matrix} n_{1;1} & n_{1;2} \end{matrix} \\ \begin{matrix} m_{1;1} \\ m_{1;2} \end{matrix} & \begin{pmatrix} A_{1;1,1} & A_{1;1,2} \\ A_{1;2,1} & A_{1;2,2} \end{pmatrix} \end{matrix},$$

where $m_{0;1} = m_{1;1} + m_{1;2}$ and $n_{0;1} = n_{1;1} + n_{1;2}$.

The HSS representation takes advantage of the low rank off-diagonal blocks. They can be factored to UBV^H using the URV decomposition. The matrix in level-one HSS representation is then written as

$$A = \begin{matrix} & \begin{matrix} n_{1;1} & n_{1;2} \end{matrix} \\ \begin{matrix} m_{1;1} \\ m_{1;2} \end{matrix} & \begin{pmatrix} D_{1;1} & U_{1;1}B_{1;1,2}V_{1;2}^H \\ U_{1;2}B_{1;2,1}V_{1;1}^H & D_{1;2} \end{pmatrix} \end{matrix}.$$

The subscripts of the variables will now be explained. The first subscript of each variable shows the level. The second subscript of a variable will depend on what level the variable occurs in. Given a particular level, the second subscript indicates the position of the block from left to right for n , V and D , and the position of the block from top to bottom for m , U and D . The matrix B has two indices indicating its position in a certain level, where the first indicates the position from top to bottom and the second from left to right.

For a multiplication or a solver it is more convenient to display the same information in an HSS tree. The two level-one HSS partition trees are displayed in Figure 1. Clearly both have the same spine. Therefore they can be super imposed and then decorated. We call this construct an HSS tree. The level-one HSS tree can be seen in Figure 2.

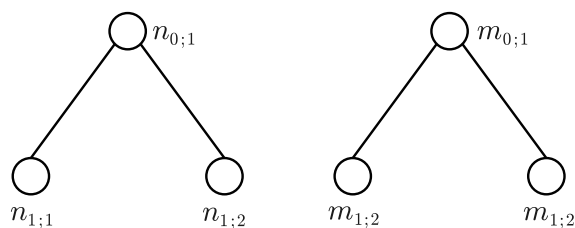


Figure 1: The level-one HSS horizontal partition tree $m_{0;1} = m_{1;1} + m_{1;2}$ on the right, and on the left the level-one HSS vertical partition tree $n_{0;1} = n_{1;1} + n_{1;2}$.

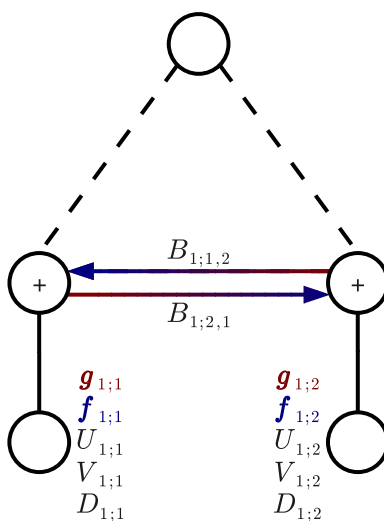


Figure 2: The HSS tree holding all the information to construct a level-one HSS representation.

The partition can be done iteratively. The second level HSS representa-

tion is

$$A = \begin{pmatrix} m_{2;1} \\ m_{2;2} \\ m_{2;3} \\ m_{2;4} \end{pmatrix} \begin{pmatrix} \begin{pmatrix} n_{2;1} & n_{2;2} \\ A_{2;1,1} & A_{2;1,2} \\ A_{2;2,1} & A_{2;2,2} \end{pmatrix} & \begin{pmatrix} n_{2;3} & n_{2;4} \\ A_{1;1,2} \\ A_{1;2,1} \end{pmatrix} \\ \begin{pmatrix} A_{2;3,3} & A_{2;3,4} \\ A_{2;4,3} & A_{2;4,4} \end{pmatrix} \end{pmatrix}.$$

where $m_{1;i} = m_{2;2i-1} + m_{2;2i}$ and $n_{1;i} = n_{2;2i-1} + n_{2;2i}$ for $i = 1, 2$. Here again the URV decomposition takes advantage of the low rank in all the off diagonal elements. This leads to the following matrix

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

where some U's and V's depend on each other. The connection is through the following equations

$$U_{1;i} = \begin{pmatrix} U_{2;2i-1}R_{2;2i-1} \\ U_{2;2i}R_{2;2i} \end{pmatrix}, \quad i = 1, 2, \quad (1)$$

$$V_{1;i} = \begin{pmatrix} V_{2;2i-1}W_{2;2i-1} \\ V_{2;2i}W_{2;2i} \end{pmatrix}, \quad i = 1, 2. \quad (2)$$

The second level HSS tree is displayed in Figure 3.

1.2 Modular Tree Representation

A good way of implementing the fast multiplication is to do all the operations on a HSS tree. Having a modular HSS tree representation of the matrix helps to keep the complexity of the problem low. With a modular HSS tree representation it is possible to write the whole algorithm as a simple iteration over the different tree modules.

Each HSS tree consists of only two different modules (branching module and leaf module) displayed in Figure 4. For example the level-two HSS tree of Figure 3 consists of three such branching modules, and 4 leaf modules. Special attention has to be paid to the top node, since it does not need W 's nor R 's.

Note that the level-two HSS tree of Figure 3 is a balanced HSS tree. An unbalanced HSS tree can be constructed as well. For example, by combining two branching modules and three leaf modules. In the matrix representation this would mean that just one of the diagonal blocks of the level-one HSS representation will be partitioned into submatrices. Figure 5 illustrates this unbalanced HSS tree, and the corresponding matrix. All the operations, for example matrix vector multiplication, and the solver, can handle unbalanced trees.

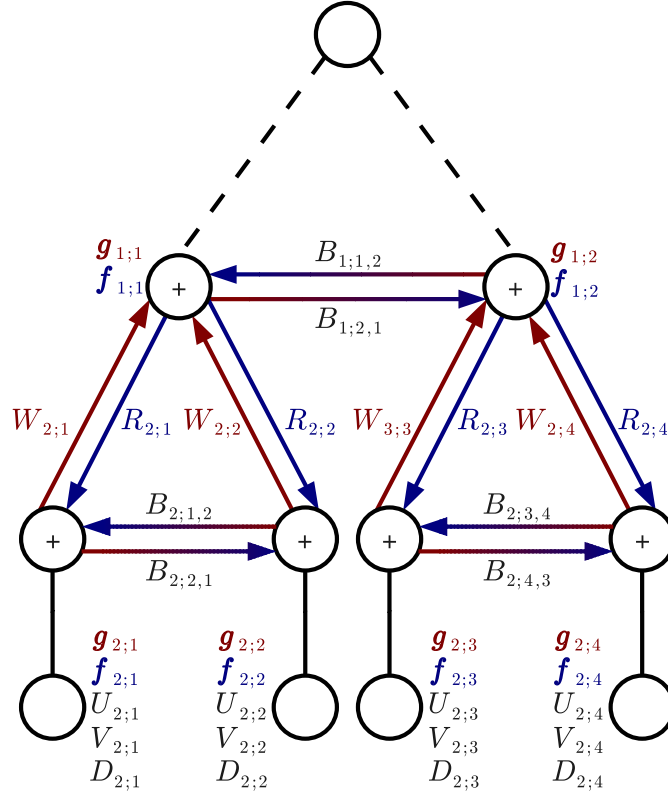


Figure 3: A HSS tree holding all the information to construct a level-two HSS representation.

1.3 HSS Toolbox

A MATLAB Toolbox [3] for matrices in HSS form was used in this thesis. Some functions needed in this thesis were not yet implemented in the Toolbox. Only those functions are explained and implemented during this thesis.

2 Algorithms for Matrices in HSS form

2.1 HSS Inverse of a Triangular Matrix

2.1.1 Introduction to Block Inverse of a Triangular Matrix

For a square lower triangular block matrix A we find its inverse B . In other words $AA^{-1} = AB = I$ holds. The block matrices A and B are given as follows:

$$A = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

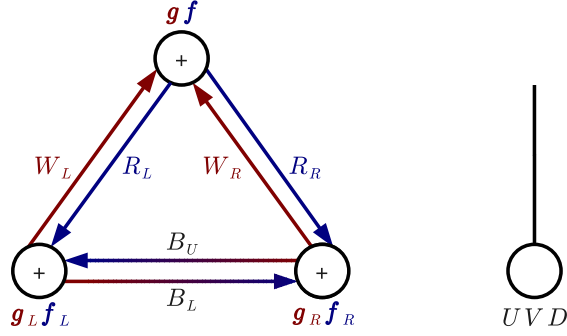


Figure 4: Branching module on the left and leaf module on the right are the two modules to build all HSS trees used here.

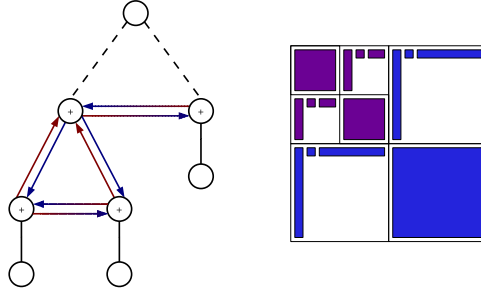


Figure 5: Illustration of an unbalanced HSS tree build with two branching modules and three leaf modules, and its representation in matrix form.

The entries of B can be identified by multiplying

$$\begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix}.$$

The equation $A_{11}B_{11} = I$ implies

$$B_{11} = A_{11}^{-1},$$

where $A_{11}B_{12} = 0$ implies

$$B_{12} = 0,$$

and with this result $A_{21}B_{12} + A_{22}B_{22} = A_{22}B_{22} = I$ implies

$$B_{22} = A_{22}^{-1},$$

and $A_{21}B_{11} + A_{22}B_{21} = 0$ implies

$$B_{21} = -A_{22}^{-1}A_{21}B_{11} = -A_{22}^{-1}A_{21}A_{11}^{-1}.$$

Therefore the inverse of the lower triangular block matrix A is given by

$$A^{-1} = \begin{pmatrix} A_{11}^{-1} & 0 \\ -A_{22}^{-1}A_{21}A_{11}^{-1} & A_{22}^{-1} \end{pmatrix}.$$

This result is used in the derivation of the inverse of a triangular matrix in HSS form.

2.1.2 Derivation

We try to find the recursion for the matrix inversion

$$A_{0;0,0}$$

where A is in HSS form. Now we look at the first level representation

$$\begin{pmatrix} A_{1;0,0} & U_{1;0}B_{1;0,1}V_{1;1}^H = 0 \\ U_{1;1}B_{1;1,0}V_{1;0}^H & A_{1;1,1} \end{pmatrix}.$$

By the magic of recursion, we assume that we computed the matrix inversion of the $(0,0)$ and $(1,1)$ terms:

$$A_{1;0,0}^{-1} = A_{1;0,0}(inv)$$

$$A_{1;1,1}^{-1} = A_{1;1,1}(inv)$$

We then have the block matrix inversion

$$\begin{aligned} & \begin{pmatrix} A_{1;0,0} & U_{1;0}B_{1;0,1}V_{1;1}^H = 0 \\ U_{1;1}B_{1;1,0}V_{1;0}^H & A_{1;1,1} \end{pmatrix}^{-1} \\ &= \begin{pmatrix} A_{1;0,0}(inv) & 0 \\ -A_{1;1,1}(inv)U_{1;1}B_{1;1,0}V_{1;0}^H A_{1;0,0}(inv) & A_{1;1,1}(inv) \end{pmatrix}. \end{aligned}$$

Now, we need a recursion to compute

$$A_{1;1,1}(inv)U_{1;1}, \text{ and } A_{1;0,0}^H(inv)V_{1;0}.$$

Therefore, we look at the expression for $V_{k;j}(inv)$ and $U_{k;i}(inv)$. Let us start with

$$A_{1;1,1}(inv)U_{1;1} = U_{1;1}(inv),$$

We also add the definition

$$A_{1;0,0}(inv)U_{1;0} = U_{1;0}(inv).$$

though it never appears in the HSS form of A^{-1} , which is lower triangular. The fix is to assume that $B_{1;0,1} = 0$.

Since we never need $A_{1;0,0}(inv)U_{1;0}$ in the HSS form, except at the bottom level, we need to convert this into the corresponding translation operators. We go up the tree rather than down and look at

$$\begin{aligned}
U_{0;0}(inv) &= A^{-1}U_{0;0} \\
&= \begin{pmatrix} A_{1;0,0}(inv) & 0 \\ -A_{1;1,1}(inv)U_{1;1}B_{1;1,0}V_{1;0}^H A_{1;0,0}(inv) & A_{1;1,1}(inv) \end{pmatrix} \\
&\times \begin{pmatrix} U_{1;0}R_{1;0} \\ U_{1;1}R_{1;1} \end{pmatrix} \\
&= \begin{pmatrix} A_{1;0,0}(inv)U_{1;0}R_{1;0} \\ -A_{1;1,1}(inv)U_{1;1}B_{1;1,0}V_{1;0}^H A_{1;0,0}(inv)U_{1;0}R_{1;0} + A_{1;1,1}(inv)U_{1;1}R_{1;1} \end{pmatrix} \\
&= \begin{pmatrix} U_{1;0}(inv)_{1;0} \\ U_{1;1}(inv)(-B_{1;1,0}V_{1;0}^H A_{1;0,0}(inv)U_{1;0}R_{1;0} + R_{1;1}) \end{pmatrix} \\
&= \begin{pmatrix} U_{1;0}(inv)R_{1;0} \\ U_{1;1}(inv)(-B_{1;1,0}G_{1;0}R_{1;0} + R_{1;1}) \end{pmatrix}.
\end{aligned}$$

This immediately suggests the recursion

$$R_{1;0}(ing) = R_{1;0}, \text{ and } R_{1;1}(inv) = -B_{1;1,0}G_{1;0}R_{1;0} + R_{1;1}.$$

Next we need the recursions for $A_{1;0,0}^H(inv)V_{1;0} = V_{1;0}(inv)$. We also make the definition $A_{1;1,1}^H(inv)V_{1;1} = V_{1;1}(inv)$, although it never appears as A is lower triangular. We fix it by assuming that $B_{1;0,1} = 0$. The trick is again to look up the tree.

$$\begin{aligned}
V_{0;0}(inv) &= A^{-H}V_{0;0} \\
&= \begin{pmatrix} A_{1;0,0}^H(inv) & -A_{1;0,0}^H(inv)V_{1;0}B_{1;1,0}^H U_{1;1}^H A_{1;1,1}^H(inv) \\ 0 & A_{1;1,1}(inv)^H \end{pmatrix} \\
&\times \begin{pmatrix} V_{1;0}W_{1;0} \\ V_{1;1}W_{1;1} \end{pmatrix} \\
&= \begin{pmatrix} A_{1;0,0}^H(inv)V_{1;0}W_{1;0} - A_{1;0,0}^H(inv)V_{1;0}B_{1;1,0}^H U_{1;1}^H A_{1;1,1}^H(inv)V_{1;1}W_{1;1} \\ A_{1;1,1}(inv)^H V_{1;1}W_{1;1} \end{pmatrix} \\
&= \begin{pmatrix} V_{1;0}(inv)(W_{1;0} - B_{1;1,0}^H U_{1;1}^H A_{1;1,1}^H(inv)V_{1;1}W_{1;1}) \\ V_{1;1}(inv)W_{1;1} \end{pmatrix} \\
&= \begin{pmatrix} V_{1;0}(inv)(W_{1;0} - B_{1;1,0}^H G_{1;1}^H W_{1;1}) \\ V_{1;1}(inv)W_{1;1} \end{pmatrix}.
\end{aligned}$$

This suggests the recursions

$$W_{1;0}(inv) = W_{1;0} - B_{1;1,0}^H G_{1;1}^H W_{1;1}, \text{ and } W_{1;1}(inv) = W_{1;1}.$$

Now we look at the recursion for G . At the bottom level G is computed explicitly. The trick is to go up the tree looking at

$$G_{0;0} = V_{0;0}^H A_{0;0,0}(inv)U_{0;0} = V_{0;0}^H U_{0;0}(inv).$$

We can write this out explicitly as follows:

$$\begin{aligned}
G_{0;0} &= \begin{pmatrix} W_{1;0}^H V_{1;0}^H & W_{1;1}^H V_{1;1}^H \end{pmatrix} \\
&\times \begin{pmatrix} A_{1;0,0}(inv) & 0 \\ -A_{1;1,1}(inv)U_{1;1}B_{1;1,0}V_{1;0}^H A_{1;0,0}(inv) & A_{1;1,1}(inv) \end{pmatrix} \\
&\times \begin{pmatrix} U_{1;0}R_{1;0} \\ U_{1;1}R_{1;1} \end{pmatrix} \\
&= \begin{pmatrix} W_{1;0}^H V_{1;0}^H & W_{1;1}^H V_{1;1}^H \end{pmatrix} \\
&\times \begin{pmatrix} A_{1;0,0}(inv)U_{1;0}R_{1;0} \\ U_{1;1}(inv)(-B_{1;1,0}G_{1;0}R_{1;0} + R_{1;1}) \end{pmatrix} \\
&= W_{1;0}^H V_{1;0}^H A_{1;0,0}(inv)U_{1;0}R_{1;0} \\
&+ W_{1;1}^H V_{1;1}^H U_{1;1}(inv)(-B_{1;1,0}G_{1;0}R_{1;0} + R_{1;1}) \\
&= W_{1;0}^H G_{1;0}R_{1;0} + W_{1;1}^H G_{1;1}(-B_{1;1,0}G_{1;0}R_{1;0} + R_{1;1}) \\
&= W_{1;0}^H G_{1;0}R_{1;0} + W_{1;1}^H G_{1;1}R_{1;1}(inv)
\end{aligned}$$

2.1.3 The Algorithm

Iteration:

- for branching modules
 - Traverse
 - to left child node, receive \mathbf{g}_L
 - Traverse
 - to right child node, receive \mathbf{g}_R
 - $R_{L \text{ inv}} = R_L$
 - $R_{R \text{ inv}} = R_R - B_L g_L^H R_L$
 - $W_{L \text{ inv}} = W_L - B_L^H g_R W_R$
 - $W_{R \text{ inv}} = W_R$
 - $B_{L \text{ inv}} = B_L$
 - $B_{U \text{ inv}} = B_U$
 - $\mathbf{g} = R_L^H g_L W_{L \text{ inv}} + R_R^H g_R W_R$
 - return \mathbf{g} to the parent module
- for leaf modules
 - $D_{\text{chol}} = D^{-1}$
 - $U_{\text{chol}} = D^{-1}U$
 - $V_{\text{chol}} = D^{-H}V$
 - $\mathbf{g} = V^H D^{-1}U$
 - return \mathbf{g} to the parent module

2.2 HSS LU

The LU factorization algorithm for matrices in HSS form is presented in [2] and [1]. Here the algorithm is rewritten in a form suitable for the modular HSS tree representation of section 1.2.

2.2.1 The Algorithm

Start with $\mathbf{f} = 0$

Iteration:

- for branching modules
 - $\mathbf{f}_L = R_L \mathbf{f} W_L^H$
 - Traverse
 - to left child node, receive \mathbf{g}_L
 - $B_L(\mathcal{L}) = B_L - R_R \mathbf{f} W_L^H$
 - $B_U(\mathcal{U}) = B_U - R_L \mathbf{f} W_R^H$
 - $\mathbf{f}_R = R_R \mathbf{f} W_R^H + B_L(\mathcal{L}) \mathbf{g}_L B_U(\mathcal{U})$
 - Traverse
 - to right child node, receive \mathbf{g}_R
 - $W_R(\mathcal{L}) = W_R - B_U^H(\mathcal{U}) \mathbf{g}_L^H W_L$
 - $R_R(\mathcal{U}) = R_R - B_L(\mathcal{L}) \mathbf{g}_L R_L$
 - $\mathbf{g} = W_L^H \mathbf{g}_L R_L + W_R^H(\mathcal{L}) \mathbf{g}_R R_R(\mathcal{U})$
 - $R_L(\mathcal{L}) = R_L(\mathcal{U}) = R_L$
 - $R_R(\mathcal{U}) = R_R$
 - $W_L(\mathcal{L}) = W_L(\mathcal{U}) = W_L$
 - $W_R(\mathcal{U}) = W_R$
 - $B_L(\mathcal{U}) = 0$
 - $B_U(\mathcal{L}) = 0$
 - return \mathbf{g} to the parent module
- for leaf modules
 - $D(\mathcal{L}) = \mathcal{L}$
 - $D(\mathcal{U}) = \mathcal{U}$
 - $U(\mathcal{L}) = U$
 - $U(\mathcal{U}) = \mathcal{L}^{-1} U$
 - $V(\mathcal{L}) = \mathcal{U}^{-1} V$
 - $\mathbf{g} = V^H D^{-1} U = V(\mathcal{L})^H U(\mathcal{U})$
 - return \mathbf{g} to the parent module

3 Introduction to Sign Function Iteration

We consider the Lyapunov matrix equation with factored right-hand side:

$$AX + XA^T = -BB^T,$$

where $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$ are the coefficient matrices, and $X \in \mathbb{R}^{n \times n}$ is the desired solution. We use the sign function iteration solver proposed in [4]. For a c-stable matrix A they suggest to use a Newton iteration with

$$M = \begin{bmatrix} A^T & 0 \\ BB^T & -A \end{bmatrix}$$

where the iteration convergence to

$$M_\infty = \lim_{k \rightarrow \infty} = \begin{bmatrix} -I_n & 0 \\ X & I_n \end{bmatrix}.$$

If the rank of X is not small enough this matrix M_∞ is not suitable for an HSS representation. Using two iterations with two $n \times n$ matrices leads to the following algorithm called GECLNW:

$$\begin{aligned} A_0 &\leftarrow A, Q_0 \leftarrow BB^T \\ \text{for } k &= 0, 1, 2, \dots \text{ until convergence} \\ A_{k+1} &\leftarrow \frac{1}{2c_k}(A_k + c_k^2 A_k^{-1}) \\ Q_{k+1} &\leftarrow \frac{1}{2c_k}(Q_k + c_k^2 A_k^{-1} Q_k A_k^{-T}) \end{aligned}$$

In [4] the scalar c_k is chosen with the Euclidian (2-norm) scaling:

$$c_k \leftarrow \sqrt{\frac{\|A\|_2}{\|A^{-1}\|_2}} \quad (3)$$

To avoid the expensive computation of the matrix 2-norm the approximation $\|A\|_2 \approx \sqrt{\|A\|_1 \|A\|_\infty}$ could be used.

For the Lyapunov equation with factored right-hand side the solution X is positive semidefinite. Therefore the factor $LL^T = X$ exists, where $\text{rank}(X) = \text{rank}(L)$. If the factor L is required instead of the explicit solution X the GECLNC algorithm can be used:

$$\begin{aligned} A_0 &\leftarrow A, B_0 \leftarrow B \\ \text{for } k &= 0, 1, 2, \dots \text{ until convergence} \\ A_{k+1} &\leftarrow \frac{1}{2c_k}(A_k + c_k^2 A_k^{-1}) \\ B_{k+1} &\leftarrow \frac{1}{\sqrt{2c_k}}(B_k, c_k A_k^{-1} B_k) \end{aligned}$$

where $\lim_{k \rightarrow \infty} B_k B_k^T = X$, in other words B_∞ is the factor of X . Note the size of B_k doubles in each iteration. This problem can be reduced by computing the rank revealing QR (RRQR) factorization [5] after each iteration step. This reduces the size of B significantly. The RRQR factorization of B_{k+1}^T is given by

$$B_{k+1}^T \Pi_{k+1} = U_{k+1} \begin{bmatrix} R_{k+1} \\ 0 \end{bmatrix},$$

where Π_{k+1} is the permutation matrix, U_{k+1} is an orthogonal matrix, and R_{k+1} is an upper triangular matrix. By truncating negligible entries of R_{k+1} , the matrix size can often be further reduced. By

$$\begin{aligned} B_{k+1} B_{k+1}^T &= \Pi_{k+1} R_{k+1}^T U_{k+1}^T U_{k+1} R_{k+1} \Pi_{k+1}^T \\ &= (\Pi_{k+1} R_{k+1}^T) (R_{k+1} \Pi_{k+1}^T) \\ &= R_{k+1}^T R_{k+1} \end{aligned}$$

the B_{k+1} can be replaced by the smaller R_{k+1}^T .

3.1 Iterative refinement

The solution computed in section 3 may be less accurate than desired. Section 2.3 of [4] propose an iterative refinement to improve the accuracy of the solution. This iterative refinement would work as well with the sign function iteration in HSS arithmetic.

4 Sign Function Iteration in HSS Arithmetic

4.1 Implementation of GECLNC

Here we first discuss two key points in the computation of the GECLNC algorithm in HSS arithmetic.

4.1.1 Computation of the Inverse in HSS Arithmetic

One way of computing the exact inverse of a matrix A is to factor A into its LU factors. Then both factors can be inverted and multiplied.

```
[L,U] = hss_lu(A);
L_inv = hss_inv_L(L);
U_inv = hss_transpose(hss_inv_L(hss_transpose(U)));
A_inv = hss_x_hss(U_inv,L_inv);
```

This is probably easier than finding an algorithm which directly inverts the matrix in HSS form. The disadvantage is that the current algorithm for the multiplication of $U^{-1}L^{-1}$ almost double the memory usage of the matrix in HSS form. This can be fixed with a so called compression algorithm, which is not yet implemented.

4.1.2 Computation of c_k

According to equation (3) the approximative 2-norm of A and A^{-1} is used to compute the scalar c_k . In section 4.1.1 we saw already how to compute the inverse of a matrix in HSS form. Therefore this section shows how to compute the 2-norm of a matrix A in HSS form.

Matlab provides the function `normest1` which estimate the 1-norm of a matrix by the block 1-norm power method. This function can be used with a function-handle which provides the multiplication AX , $A^T X$ and the information that A is real and the size of A . Since the multiplication of A with a matrix X and the transpose of A is in the HSS Matlab Toolbox this function-handle can be written. The function `hss_normest1(A)` estimates the 1-norm of the matrix in HSS form.

The 2-norm $\|A\|_2$ can then be estimated with $\sqrt{\|A\|_1 \|A\|_\infty}$. With $\|A\|_\infty = \|A^T\|_1$ we can compute $\|A\|_2 \approx \sqrt{\|A\|_1 \|A^T\|_1}$

4.1.3 Computation of the GECLNC Algorithm

The rest of the implementation of the GECLNC algorithm is straight forward. The multiplication of a matrix in HSS form with a scalar, the addition and multiplication of two matrices in HSS form, and a solver are available as a matlab function. They can be used to compute the A and B after one GECLNC step. The compression of B is done with an ordinary RRQR algorithm since B is not in HSS form.

5 Experimental Results

5.1 Test Setup

The tests run on the following test setup:

- Software
 - Operating system: 2.6.32-22-generic-Ubuntu
 - Matlab: 7.10.0.499 (R2010a)
- Hardware
 - Ram: 7.7 GiB
 - Swap space: 3.8 GiB
 - CPU: Intel(R) Pentium(R) Core 2 Duo E4800 @ 3.0GHz

5.2 Tuning

As in [1] the solver has a tuning parameter called `maxLeafSize`. This section shows how this parameter should be chosen. The `maxLeafSize` determine when the program stops partitioning the matrix into submatrices. As soon as a submatrix has less than `maxLeafSize` columns and rows, the submatrix is used as a leaf node. Small values for `maxLeafSize` leads to many levels in the HSS representation, and therefore many iterations in the algorithms used. Large `maxLeafSize` leads to fewer levels, but more memory usage, since the HSS form compresses the matrix. Therefore a reasonable `maxLeafSize` has to be determined. One good way is to solve the same Lyapunov equation with different `maxLeafSize`, and measure the computation time. This is shown in Figure 6, where three iterations with a matrixsize of 10^5 are computed. With this computer `maxLeafSize` is best chosen between 50 and 200.

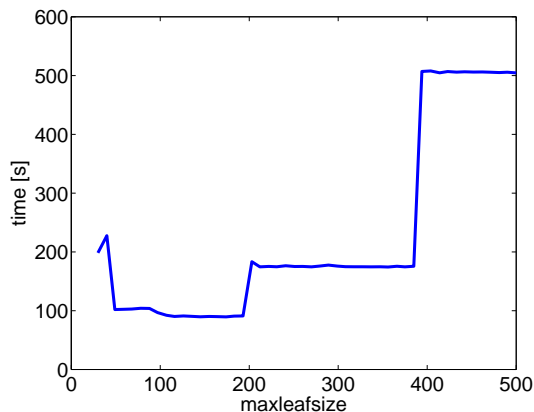


Figure 6: The optimal `maxLeafSize` depends on the computer the code is running on.

The memory usage changes with `maxLeafSize`. This is shown in Figure 7. As `maxLeafSize` decreases the memory usage decreases as well, till a memory optimum is found. For computers with small memory it is therefore good to choose `maxLeafSize` carefully.

5.3 Convergence

In Figure 8 the convergence of the relative error is measured with the Laplacian as A and a random vector as B . The size of the matrix A is 10^4 .

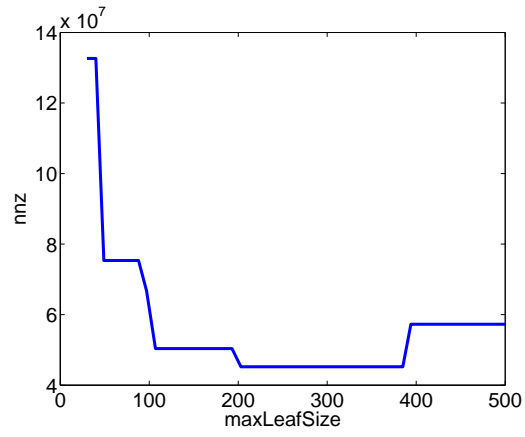


Figure 7: The number of non zeros in the HSS tree after 2 GECLNC steps depends on the parameter `maxLeafSize`.

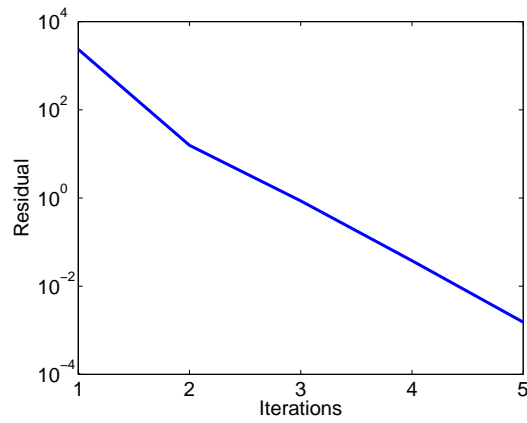


Figure 8: A loglog plot of the relative error for the Solution of the Lyapunov equation showing the convergence.

5.4 Speed

This Lyapunov equation solver should be linear in time for the size of the matrix A . We vary the size of the matrix A , to verify that the time cost increases linearly.

Figure 9 shows the time used to compute the Lyapunov equation for a different size of the matrix A . There three iterations are computed with the parameter `maxLeafSize = 150`. The total time evolves almost linearly with the matrix size, as expected from theory.

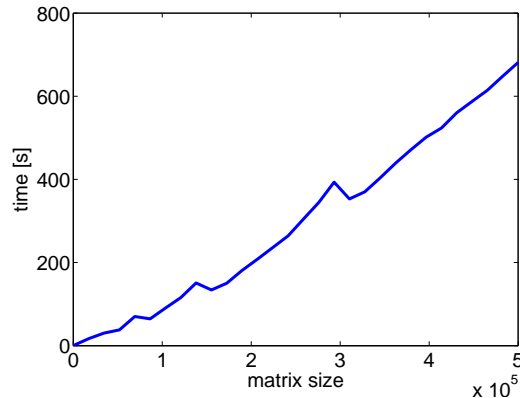


Figure 9: Time used for the Lyapunov solver dependent on the matrix size of A , with three iterations and `maxLeafSize = 150`.

6 Summary

During this thesis a solver for Lyapunov equations

$$AX + XA^T = -BB^T,$$

where A is in HSS form was implemented and the results documented. This was achieved by implementing the GTECLNC algorithms of [4] for matrices in HSS form. Therefore some algorithms for matrices in HSS form were used from the HSS Matlab Toolbox [3], others were derived and implemented during the thesis.

6.1 Outlook

Here we discuss some useful improvements for the Lyapunov equation solver:

- The rank used in the HSS representation may be bigger than necessary for a given accuracy. In this case the so called compression

algorithm could find a better HSS form of the matrix. Applying this algorithm after each GECLNC step would decrease memory usage and computation time, especially for many iterations.

- A direct inverse of a matrix in HSS form could be faster than the current algorithm with the LU factors as intermediate step. This algorithm should be derived.
- The code is so far not very efficient. There is still optimization potential in terms of computation time.

References

- [1] S. Pauli: Fast Algorithms and Applications for Matrices in HSS Form: master thesis, 2010
- [2] W. Lyons: Fast Algorithms with Applications to PDEs: PHD thesis, 2005
- [3] S. Pauli, K. R. Jayaraman, and S. Chandrasekaran, [MATLAB Toolbox for Hierarchically Semiseparable Representations and MSN Interpolation], <http://scg.ece.ucsb.edu> (2009).
- [4] P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortí, and A. Remón: A Mixed-Precision Algorithm for the Solution of Lyapunov Equations on Hybrid CPU-GPU Platforms: Research Report No. 2009-40, Seminar für Angewandte Mathematik, ETH Zürich, December 2009.
- [5] G. Golub, C. V. Loan: Matrix Computations: 3rd Edition, The Johns Hopkins University Press, Baltimore, 1996