An Evaluation of Two Algorithms for Hierarchically Semiseparable Matrices

Scholarly Paper, Computer Science Department, University of Maryland, Spring 2011
Brianna Satinoff
Advisor: Dianne O'Leary

Abstract

The hierarchically semiseparable (HSS) representation is a method introduced by Chandrasekaran et al. to divide sparse matrices into a hierarchical block structure. They used this representation for a Cholesky factorization algorithm that, for certain types of matrices, was faster than the standard LAPACK one. In this paper I re-implement this factorization, as well as their algorithm for putting a matrix in HSS form. I then evaluate this implementation on a set of matrices from a variety of sources as well as some randomly generated matrices. The experiments partially confirm the authors' conclusions about the asymptotic complexity of the HSS algorithms. They also confirm that many real-world matrices have the property, called the ``low-rank property'', that makes them conducive to HSS representation. However, they fail to find an existing Cholesky implementation that forms a good standard comparison.

1. Introduction

Direct solution of sparse linear systems is useful because it produces an exact answer, and because it can be quickly repeated with different right-hand sides. It is often desirable to start by factoring the matrix using the LU algorithm or, for symmetric positive definite matrices, the Cholesky algorithm. In this paper, we will focus on symmetric positive definite matrices and the Cholesky factorization.

When directly calculating the Cholesky factorization of a sparse matrix, the lower-triangular factor often has many more nonzero terms than the original matrix. This "fill-in" is undesirable because it requires extra storage, and because solving the linear system takes longer. Thus we want to rearrange the matrix's rows and columns to minimize fill-in. At the same time, we want to minimize the time that the rearrangement takes.

No practical method finds the best rearrangement for every matrix, but there are several well-known heuristics. The examples described here all treat the matrix as a graph, with a vertex for each row or column and an edge for each nonzero entry. That is, if the matrix has a nonzero entry at row 3 and column 5, for example, an edge connects nodes 3 and 5. The minimum degree algorithm simply picks the node with minimum degree at each step. Cuthill-McKee starts with a minimum-degree node and then does a breadth-first traversal. Nested dissection algorithms divide the graph hierarchically instead of considering each node on its own. In each step, they split the nodes between two halves and a separator, so that the two halves aren't connected by any edges. They repeat this recursively until some stopping point, then calculate the ordering from the bottom up.

The hierarchically semiseparable (HSS) representation is another way to split a sparse matrix into a hierarchical structure. It relies on the matrix having off-diagonal blocks with low rank (compared to the matrix's dimension). These off-diagonal blocks can then be compressed by using rank-revealing QR and discarding all rows of R with diagonal entries less than a given tolerance. The end result is a set of HSS generator matrices that multiply out to approximate the blocks of the original matrix. If all the off-diagonal blocks have low rank, and the generators' ranks are not much higher, then the matrix is said to have the "low-rank property"

In (2), Chandrasekaran et al. developed algorithms for Cholesky decomposition and linear system

solving using the HSS representation. They showed that for matrices with the low-rank property, their HSS Cholesky factorization routine is faster **and** results in less fill-in than the standard one in the LAPACK library. In (1), they combined this HSS algorithm with nested dissection into an even faster algorithm, the "superfast multifrontal method". It starts with the multifrontal method of nested dissection, then approximates the resulting sub-matrices with the HSS representation.

The purpose of this paper is to evaluate the authors' results by re-implementing a subset of their HSS based algorithms. Specifically, I followed their instructions for creating HSS generators and for the generalized HSS Cholesky factorization, both from (2). The implementation was for a specific set of conditions described in section 2. I then tested my implementations with experiments on several types of real-world matrices. Although I did not implement the superfast multifrontal method in (1), conclusions can still be drawn about it because it relies on both of the implemented algorithms.

The rest of the paper is structured as follows. Implementation details and experimental methods are described in section 2. Section 3 presents a summary of the results and section 4 discusses them. The conclusion is in section 5. The actual results table is in Appendix A, since it is too long to insert into the paper. Appendix B contains some equations I derived for my implementation.

2. Methods

What was implemented?

Hierarchically semiseparable matrices can be divided according to any binary tree. Everything I implemented was for this specific tree:

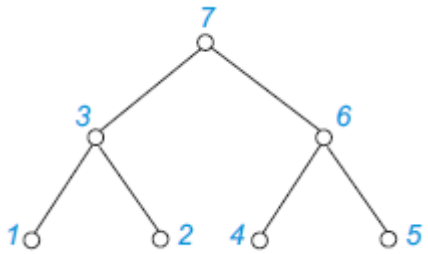

Figure 1: The binary tree used to partition the matrix.

This causes the matrix to be divided into 4 parts row-wise and 4 parts column-wise, for a total of 16 blocks with equal dimensions. So for an 100x100 matrix, each block is 25x25. The HSS algorithms operate on a set of off-diagonal “block rows”, using postordering to match the block rows to the nodes of the tree. A set of block columns could be defined instead, which would be equivalent for symmetric matrices. The above tree corresponds to the following 6 block rows:

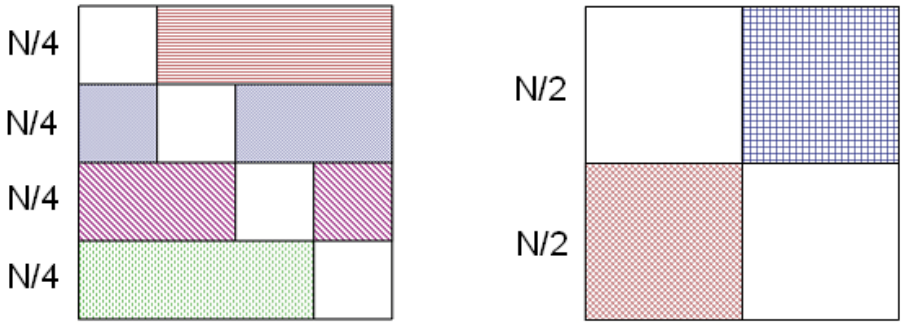

Figure 2: The 6 off-diagonal block rows used in this paper, with their relative dimensions.

First, I wrote a program to put a matrix into hierarchically semiseparable (HSS) form. This involves finding a set of "generator" matrices that combine to form the original matrix. This step is a prerequisite for any other HSS-related operation. The algorithm I implemented is described in (2) and referred to in (1). Even though the algorithm may be used for non-symmetric matrices, I implemented it for symmetric matrices only because symmetry is required for the subsequent step.

I then implemented the generalized HSS Cholesky factorization, which is presented in (2) and is part of the superfast multifrontal method in (1). The algorithm is called "generalized" because instead of returning a single lower triangular matrix and its transpose, it returns a set of lower triangular and orthogonal matrices of various sizes. They hierarchically form the original matrix, as shown on pg. 14 of (2) and in Appendix B.

Paper (1) also draws on the (non-superfast) multifrontal method for nested dissection, combining it with generalized HSS Cholesky factorization to build the superfast multifrontal method. I did not implement this.

I looked for Matlab code for the multifrontal method that I could use for comparison. I found, however, that Matlab's built-in implementation of sparse Cholesky factorization already uses nested dissection. It uses a supernodal structure, just as the proposed superfast multifrontal method does. This was much faster than my implementation, as described in the Results section.

Experiments

I tested the algorithm on 3 different types of matrices:

- Randomly generated sparse matrices, which were generated using the Matlab sprandsym command. This command (with the options I used) outputs a symmetric positive definite matrix with a given size, inverse condition number, and proportion of nonzero elements. Some were altered to set a specific HSS rank, and some were not.
- A subset of the symmetric positive definite matrices on the University of Florida sparse matrix collection, ranging in size from 136 to 5488. Most, but not all, have very high condition numbers. I tried to choose a few matrices from each group.
- A set of 9 matrices formed by solving elliptic PDE's using finite elements in Matlab's PDE toolbox. The variations were the number of mesh refinements (1, 2 or 3) and the boundary shape: square, circular, or L-shaped. All six PDEs had the following properties:
  - PDE: $-\Delta x = 10$ (a form of the Poisson equation; the default in pdetool)

- Boundary conditions: 0 everywhere (the default in pdetool)

The tests consisted of the following steps for each matrix:

1) Put the matrix in HSS form, using a tolerance of 1E-5. Output the HSS representation rank.
    1. Note: The HSS representation indicates how "compressed" the HSS representation is; it is the maximum rank of the HSS generator matrices. This may differ from the HSS rank, which is just the maximum rank of the off-diagonal blocks in the **original** matrix. The maximum for both is ½ the dimension of the matrix. Both must be low for the matrix to have the low-rank property.
2) Run the generalized HSS Cholesky factorization. Output:
    1. The *total* number of nonzero below-diagonal elements in all of the lower-triangular generalized Cholesky factors. Use a tolerance of 1E-5 when determining if an element is zero.
    2. The *maximum* number of nonzero below-diagonal elements in these factors.
    3. The total time taken for steps 1 and 2.
3) Run the built-in Cholesky factorization on the matrix, in both sparse and full form. Output the time taken and the number of nonzero below-diagonal elements. For consistency with the HSS results, use a tolerance of 1E-5 when determining if an element is zero.
4) Repeat steps 1-3 with a tolerance of 1E-10 instead of 1E-5.

I tested on the matrices from pdetool and from the UF website. Descriptions of these matrices are part of the Table of Results.

I also tested on two types of randomly generated matrices, since the authors achieved good results on random matrices. The specifications were:

- Generated using Matlab's SPRANDSYM
- Size: 1024 x 1024
- Proportion of nonzeros: 0.1

In addition, I altered some of these matrices to obtain a specific upper bound on HSS rank. Given some rank p, the goal was to give each of the 6 off-diagonal block rows a rank of at most p. To accomplish this, each block of dimension N/4 was assigned a rank as follows:

N/4	-	p/2	0	p/2
N/4	p/2	-	p/2	0
N/4	0	p/2	-	p/2
N/4	p/2	0	p/2	-

Figure 3: The rank assigned to each block of the random matrix, to achieve an overall HSS rank p.

The blocks with rank 0 were simply zeroed out. For each above-diagonal block with rank p/2, all but p/2 randomly selected columns were zeroed out. To maintain symmetry, the corresponding below-

diagonal block is just the transpose of this block. Finally, if the resulting matrix may not be positive definite according to Gershgorin's circle theorem, a positive multiple of the identity matrix was added so that it was.

Although this process gives the matrices some non-random structure, it should still be more experimentally reliable than using a collection of non-random matrices.

Correctness of Implementation

My experiments included methods to verify that my implementation was correct. An equation in (2) shows how the HSS generators of a 4x4 block matrix should multiply out to form each block of the original matrix. I checked the accuracy of the generators by substituting them into this equation. The 2-norm of the error for each block was always less than the supplied tolerance, and in most tests was near machine precision.

For the HSS Cholesky factorization, another equation in (2) shows how the generalized Cholesky factors should multiply out to the original matrix. This equation only applies to 2x2 block matrices, so I derive a generalization for 4x4 block matrices in Appendx B. Again, I checked the accuracy of the generalized Cholesky factors by substituting them into this equation. The 2-norm of the error was always close to the above error from forming the generators, so the factorization itself added negligible error.

3. Results

The following tables show correlation coefficients between various pairs of measurements. The full results for the non-random matrices are also in Appendix A.

Legend

N = dimension of matrix

p = HSS rank of matrix (the maximum rank of the 6 off-diagonal blocks)

Correlations for random matrices

I am considering the random matrices separately because they all have the same dimension, so the effects of p can be analyzed without considering N. In addition, I am considering only the HSS rank, not the HSS representation rank, because they were identical for the random matrices and nearly identical for the non-random ones.

Dependent variable	Correlation with:	
	p	p^2
Time to form HSS generators	0.9419	0.9053
Time for HSS Cholesky factorization	0.9247	0.8656
(Time for HSS Cholesky factorization) / (time for CHOL)	0.9284	0.8870
(Time for HSS Cholesky factorization) / (time for CHOLMOD)	0.9346	0.8956

(time for CHOLMOD) / (time for CHOL)	-0.0080	-0.0348
Maximum nonzeros in HSS Cholesky factors	0.9489	0.9883
Total nonzeros in HSS Cholesky factors	0.9646	0.9231
Total nonzeros in CHOL result	0.9942	0.9850
(Maximum nonzeros) / (total nonzeros) in HSS Cholesky factors	0.9374	0.9818
(Maximum nonzeros in HSS Cholesky factorization) / (total nonzeros in CHOL)	0.7615	0.8348
(Total nonzeros in HSS Cholesky factorization) / (total nonzeros in CHOL)	-0.9346	-0.8572

Correlations for non-random matrices

These matrices allow us to consider the effect of N in addition to the other factors.

Independent variable	Dependent variable	Correlation
N	Time to form HSS generators	0.6361
N^2	Time to form HSS generators	0.4800
N * p	Time to form HSS generators	0.8277
N * p^2	Time to form HSS generators	0.9292
N	Time for HSS Cholesky factorization	0.6587
N^2	Time for HSS Cholesky factorization	0.5136
N * p	Time for HSS Cholesky factorization	0.8298
N * p^2	Time for HSS Cholesky factorization	0.9108

Other results

All of the purely random matrices – the ones where the HSS rank was not explicitly set – had HSS ranks very close to the maximum of N/2. The HSS ranks ranged from 433 to 454, with N = 1024.

The number of below-diagonal nonzeros in the generalized Cholesky factors can be measured in two ways: the *total* number of nonzeros in all the factors, and the *maximum* number in any factor. The standard Cholesky factorization only produces one factor. All of the matrices, regardless of HSS rank, had more *total* nonzeros than the standard Cholesky factor. Some of the matrices, however, had fewer *maximum* nonzeros than the standard factor. All of the matrices with this property had relatively low HSS rank, but the converse is not true. An example is matrix 1138_bus (from the UF collection) when using tolerance 1E-5. This matrix has dimension 1136, HSS rank 98, and a maximum of 24340 below-diagonal nonzeros per generalized factor. The standard Cholesky factor has 20847 below-diagonal nonzeros.

4. Discussion of Results

Algorithm Performance

External Comparisons

In every case, my code was several times slower than Matlab's built-in sparse Cholesky (CHOLMOD) and even its full Cholesky (CHOL) functions. This doesn't necessarily mean that my code has the wrong big-O performance, but simply that the constant factor is very large. Several factors may have caused this slowness. First, much of the logic in my code is in Matlab, while both CHOLMOD and CHOL are entirely in C. Matlab is an interpreted language, while C gets a performance advantage by being compiled into machine code. Also, I used various Matlab constructs such as cells without really knowing how efficient their underlying implementation is. Choosing certain constructs over others might turn out to improve the code's performance.

I tried to create a fairer comparison by testing my code against a simple, pure-Matlab, no-optimization Cholesky implementation. [1] However, that was many times slower than my code even on matrices with near-maximum HSS rank, probably because my code does use some built-in functions such as QR..

The conclusion is that external comparisons don't work in this case. My implementation is a heterogeneous mix of interpreted Matlab and calls to built-in, compiled functions. It can't fairly be compared with either all-compiled (such as CHOL and CHOLMOD) or all-Matlab code. A future task would be to re-implement the algorithms in a single language, using a matrix library written in the same language. The authors already did this with Fortran and the LAPACK library. There are also C and Java numerical libraries. [2]

Internal Comparisons

The authors claim in (2) that forming the HSS generators should take O(N^2) time, where N is the dimension of the matrix. This would mean a low correlation between p and this runtime for the random matrices, which all had the same size. However, a strong positive correlation was found, so the complexity of my implementation does include a factor of p. Furthermore, for the non-random matrices, the correlations between this runtime and N, N*p, and N*p^2 were all stronger than N^2. This can be explained by noting some constraints that the authors set but I could not. In their big-O proof, they assume that m (the block size) is O(p). In the experiments, they specifically set m = 2*p. I could not do this, because my implementation was only for a particular block size (m = N/4). Another possibility is that the constant factors that slowed down my code's performance had most of their impact on the N, N*p, and N*p^2 terms.

The authors also claim in (2) that generalized HSS Cholesky factorization should take O(N*p^2) time, where p is the matrix's HSS rank and N is its dimension. There was indeed a strong correlation – with p^2 for the random matrices and N*p^2 for the nonrandom ones. However, the random matrices gave a higher correlation with p. Again, the difference between the observed and expected relationships could be explained by the same factors as above.

---

1 The Matlab code was found at http://www.ece.uwaterloo.ca/~dwharder/NumericalAnalysis/04LinearAlgebra/cholesky/.

2 A C example is the GNU Scientific Library, at http://www.gnu.org/software/gsl/. A Java example is Java Numerics, at http://math.nist.gov/javanumerics/.

The difference between my results and theirs raises a set of interesting questions. How would the efficiency of the HSS Cholesky factorization change if m << 2p or m >> 2p? Would the runtime increase in both cases, or would splitting the matrix into "too many" blocks have no effect? The experiments in (1) provide a hint. They were all run on the same class of matrices, but with different HSS block sizes. The runtime did bottom out at a certain level and then go back up. However, this result was for the combination of multifrontal nested dissection and HSS Cholesky factorization, so it might not apply for the HSS method alone.

Nonzeros in Algorithm Output

Performance is not the only goal of a Cholesky factorization algorithm; minimizing the number of below-diagonal nonzeros in the factors is also important. However, this algorithm outputs multiple lower-triangular factors, so there are two counts to consider: the *total* number of below-diagonal nonzeros in all the factors, and the *maximum* such number in any factor. Both increased as the HSS rank increased, as expected. However, the relationship between them also varied. For matrices with near-maximum HSS rank, the maximum count was close to the total count, so most of the below-diagonal nonzeros were concentrated in a single factor. For lower-HSS-rank matrices, the counts were further apart, so the nonzeros were more evenly distributed. The positive correlation between p and the (maximum count / total count) ratio for the random matrices confirms this relationship.

I'm not certain whether the total count or the maximum count is a better indicator of the usefulness of the factorization. My understanding is that HSS applications use each generalized factor by itself, rather than combining them together. In these cases, a low maximum count might be important.

Patterns in HSS Rank

As stated by the authors and verified in my experiments, a matrix's HSS rank needs to be small relative to its dimension in order for the HSS representation to be useful. After all, using the authors' rule of thumb that m = 2p (where m is the block size and p is the HSS rank), an HSS rank of close to N/2 means that the block size should be N. That is, the matrix should not be split up at all! So an important question is, what type of matrices have the low-rank property?

Many of the real-world matrices from the UF library had low HSS rank, including 1138_bus (size 1138, HSS rank 98) and crystm01 (size 4875, HSS rank 390). (I would expect more dramatic results if I had implemented the HSS formation algorithm with more than 16 blocks.) However, the random matrices I generated all had HSS ranks fairly close to the maximum of N/2. So the low-rank property is not typical for sparse matrices in general.

The authors also state that the low-rank property often occurs in matrices that come from solving elliptic PDEs using finite elements. The set of such matrices that I tested on only showed this property to a limited degree. With 1 mesh refinement, the HSS ranks ranged from 0.68 to 0.78 of the maximum – close enough to make the HSS representation worthless. With 3 refiments, they were all about 0.50 of the maximum – a small improvement, but still far from the “low-rank” realm. The matrices might need more refinements to really show the low-rank property, but the close proximity of the HSS ranks for the 3-refinement matrices suggests some type of asymptote. Perhaps only certain classes of elliptic PDE's generate matrices with the low-rank property.

For the matrices with HSS rank near the maximum of (size)/2 – the random matrices, the matrices from pdetool, Trefethen200_b, and Trefethen_300 – the built-in sparse Cholesky factorization took about as

long or longer than the full one. For the other matrices, the built-in sparse factorization took much less time. This suggests that matrices with low HSS rank also have some property that makes the CHOLMOD algorithm more efficient.

Limitations of Work

The following limitations refer specifically to observations I made about my implementation.

When testing on small random matrices, I encountered the situation where one of the off-diagonal blocks is all zeros. The HSS generators couldn't be formed in that case. I assume that matrices generated from real-world applications generally do not have this property.

The HSS Cholesky factorization failed on four of the matrices with high condition number when using a tolerance of 1E-5. An intermediate matrix was supposed to be positive definite and had lost that property. In all but one of the matrices where this happened, using a tolerance of 1E-10 fixed the problem. This makes sense: a high condition number means that small changes to a matrix may totally alter its properties. In this case, the small change was eliminating the singular values that were less than 1E-5 when creating the HSS generators.

5. Conclusions

Overall, the experiments were a mixed success. They confirmed the authors' conclusions about asymptotic complexity for the HSS generator formation algorithm. They supported, but did not definitely confirm, this asymptotic complexity for the HSS Cholesky factorization. However, the attempts to find an external standard of comparison were failures.

The experiments also corroborated the authors' statement that many matrices from physical and mathematical applications have the low-rank property. The UF matrices showed strong results here, even though the elliptic PDE-based ones did not. My use of varying real-world matrices was a new addition to the authors' work; they used randomized matrices in (2) and a single class of PDE in (1).

Although I could not achieve the direct one-to-one comparisons that the authors did, I believe that my results are still sufficient to show the usefulness of the HSS representation. Several steps of future work would better duplicate or extend the authors' experiments. First, a complete re-implementation should be able to divide the matrix according to any complete binary tree, not just the small three-level one in this paper. Also, as previously mentioned, the implementation should be in a single language for better external comparisons. Finally, a more complete set of finite element problems for elliptic PDE's should be tested, to more clearly establish which ones result in the low-rank property.

List of Resources

1. Xia, J., Chandrasekaran, S., Gu, M., Li, X.S.: Superfast multifrontal method for large structured linear systems of equations. SIAM Journal on Matrix Analysis and Applications 31(3), 1382-1411 (2009). DOI 10.1137/09074543X

2. Xia, J., Chandrasekaran, S., Gu, M., Li, X.S.: Fast algorithms for hierarchically semiseparable matrices. Numerical Linear Algebra with Applications, 17: 953–976 (2010). DOI 10.1002/nla.691
3. University of Florida sparse matrix collection: http://www.cise.ufl.edu/research/sparse/matrices

Appendix A: Experimental Results

Matrix name	Size	Condition Number	HSS Rank (max = size/2)	Toler-ance	Processing time (seconds)			Number of off-diagonal nonzeros in Cholesky factors		
					HSS algorithm	Sparse Cholesky (built-in)	Full Cholesky (built-in)	Total of HSS Cholesky factors	Max of HSS Cholesky factors	Non-HSS Cholesky factor
bcsstk22	136	1.4E+5	28	1E-5	HSS Cholesky factorization failed because an intermediate matrix was not positive definite.					
				1E-10	0.0116	0.0002	0.0008	2244	573	863
lund_a	144	4.6E+5	36	1E-5	0.0117	0.0002	0.0008	5248	2637	2557
				1E-10	0.0117	0.0003	0.0007	5383	2715	2795
Trefethen_200b	196	7.1E+2	98	1E-5	0.0413	0.0023	0.0011	14379	13987	2557
				1E-10	0.0414	0.0030	0.0011	14945	14553	13495
mesh3e1	288	9.0	54	1E-5	0.0137	0.0011	0.0026	7297	1984	2947
				1E-10	0.0264	0.0011	0.0027	9657	2618	5653
mesh3em5	288	5.0	54	1E-5	0.0105	0.0012	0.0053	1200	285	270
				1E-10	0.0131	0.0012	0.0023	2733	628	914
Trefethen_300	300	2.6E+3	150	1E-5	0.1030	0.0059	0.0024	31500	30900	3923
				1E-10	0.1056	0.0067	0.0036	34575	33975	28368
mesh2em5	304	2.9E+2	88	1E-5	0.0432	0.0011	0.0038	14392	5719	2226
				1E-10	0.0732	0.0013	0.0029	20069	8150	6771
bcsstk06	420	1.2E+7	72	1E-5	0.0202	0.0013	0.0064	15023	4520	9492
				1E-10	0.0260	0.0014	0.0066	23706	7650	13501
nos5	468	2.9E+4	114	1E-5	0.0884	0.0029	0.0065	43263	18967	18633
				1E-10	0.0896	0.0060	0.0063	46292	20193	27038

Matrix name	Size	Condition Number	HSS Rank (max = size/2)	Toler-ance	Processing time (seconds)			Number of off-diagonal nonzeros in Cholesky factors		
					HSS algorithm	Sparse Cholesky (built-in)	Full Cholesky (built-in)	Total of HSS Cholesky factors	Max of HSS Cholesky factors	Non-HSS Cholesky factor
bcsstk19	816	2.8E+11	?	1E-5	HSS Cholesky factorization failed because an intermediate matrix was not positive definite.					
				1E-10	HSS Cholesky factorization failed because an intermediate matrix was not positive definite.					
bcsstk09	1080	3.1E+4	114	1E-5	0.2201	0.0060	0.0280	114808	35455	43125
				1E-10	0.2105	0.0059	0.0272	148241	44509	61382
1138_bus	1136	3.2E+6	98	1E-5	0.3181	0.0076	0.0348	57116	18141	20847
				1E-10	0.3025	0.0073	0.0309	75447	23827	36243
nasa2910	2908	1.8E+7	437	1E-5	HSS Cholesky factorization failed because an intermediate matrix was not positive definite.					
				1E-10	8.9447	0.0896	0.2619	1381350	353861	444569
nasa4704	4704	1.2E+8	533	1E-5	HSS Cholesky factorization failed because an intermediate matrix was not positive definite.					
				1E-10	11.5104	0.1729	0.8606	2316417	825071	770057
crystm01	4872	4.2E+2	390	1E-5	7.2590	0.3110	1.0494	849637	316156	33753
				1E-10	7.0759	0.3093	1.0004	1030955	380110	85026
s1rmq4m1	5488	1.7E+6	360	1E-5	14.7170	0.1937	1.3398	1582386	535819	734623
				1E-10	14.8114	0.1863	1.4299	2437364	827371	980204
Elliptic PDE w/square boundary #1	696	5.4E+3	241	1E-5	0.7246	0.0138	0.0127	133516	78597	12300
				1E-10	0.8840	0.0154	0.0148	158201	86814	29506

Matrix name	Size	Condition Number	HSS Rank (max = size/2)	Toler-ance	Processing time (seconds)			Number of off-diagonal nonzeros in Cholesky factors		
					HSS algorithm	Sparse Cholesky (built-in)	Full Cholesky (built-in)	Total of HSS Cholesky factors	Max of HSS Cholesky factors	Non-HSS Cholesky factor
Elliptic PDE w/L-shaped boundary #1	556	2.60E+003	191	1E-5	0.4308	0.0091	0.0098	84682	52217	9295
				1E-10	0.4730	0.0117	0.0099	96928	54798	19552
Elliptic PDE w/circular boundary #1	540	3.6E+3	211	1E-5	0.4381	0.0099	0.0090	90318	65110	10287
				1E-10	0.4946	0.0109	0.0078	99401	67067	22468
Elliptic PDE w/square boundary #2	2576	3.8E+4	701	1E-5	25.8294	0.1366	0.1883	1546208	610681	62499
				1E-10	30.3747	0.1339	0.2003	1843236	726770	185124
Elliptic PDE w/L-shaped boundary #2	2144	2.0E+4	615	1E-5	14.4548	0.0909	0.1238	1078433	471416	50053
				1E-10	18.1226	0.1282	0.1774	1305218	560476	136522
Elliptic PDE w/circular boundary #2	2096	2.9E+4	603	1E-5	15.4345	0.1080	0.1206	1083231	494881	54795
				1E-10	18.8469	0.1003	0.1137	1246156	543632	160383
Elliptic PDE w/square boundary #3	10144	3.0E+5	2536	For the elliptic PDE's with 3 mesh refinements, Matlab ran out of memory when calculating the HSS Cholesky factorization. So only the sizes, condition numbers, and HSS ranks are shown.						
Elliptic PDE w/L-shaped boundary #3	8416	1.6E+5	2101							
Elliptic PDE w/circular boundary #3	8256	2.3E+5	2064							

Appendix B: Equation Sheet

Equations (4.7) and (4.8) in Chandrasekaran et al's 2008 paper (2) show how to check that the generalized hierarchically semiseparable Cholesky factorization is correct for a block 2x2 matrix. I will use these equations, along with (4.1) through (4.6), to generate a similar set of equations for a block 4x4 matrix.

Call the original matrix H. Assume that we have all the results from the generalized HSS Cholesky factorization algorithm, including the D, D-hat, D-tilde, Q, and L matrices. Equations (4.1) through (4.6) show where these matrices come from. Then, starting with the 4x4 HSS form of H:

$$H=\begin{bmatrix} D_1 & U_1 B_1 U_2^T & U_1 R_1 B_3 R_4^T U_4^T & U_1 R_1 B_3 R_5^T U_5^T \\ U_2 B_1^T U_1^T & D_2 & U_2 R_2 B_3 R_4^T U_4^T & U_2 R_2 B_3 R_5^T U_5^T \\ U_4 R_4 B_6 R_1^T U_1^T & U_4 R_4 B_6 R_2^T U_2^T & D_4 & U_4 B_4 U_5^T \\ U_5 R_5 B_6 R_1^T U_1^T & U_5 R_5 B_6 R_2^T U_2^T & U_5 B_4^T U_4^T & D_5 \end{bmatrix}$$

$$Q=\begin{bmatrix} Q_1 & & & \\ & Q_2 & & \\ & & Q_3 & \\ & & & Q_4 \end{bmatrix}$$

$$H=Q\begin{bmatrix} \hat{D}_1 & \hat{U}_1 B_1 \hat{U}_2^T & \hat{U}_1 R_1 B_3 R_4^T \hat{U}_4^T & \hat{U}_1 R_1 B_3 R_5^T \hat{U}_5^T \\ \hat{U}_2 B_1^T \hat{U}_1^T & \hat{D}_2 & \hat{U}_2 R_2 B_3 R_4^T \hat{U}_4^T & \hat{U}_2 R_2 B_3 R_5^T \hat{U}_5^T \\ \hat{U}_4 R_4 B_6 R_1^T \hat{U}_1^T & \hat{U}_4 R_4 B_6 R_2^T \hat{U}_2^T & \hat{D}_4 & \hat{U}_4 B_4 \hat{U}_5^T \\ \hat{U}_5 R_5 B_6 R_1^T \hat{U}_1^T & \hat{U}_5 R_5 B_6 R_2^T \hat{U}_2^T & \hat{U}_5 B_4^T \hat{U}_4^T & \hat{D}_5 \end{bmatrix} Q^T \quad \text{from (4.1)}$$

$$\hat{L}=\begin{bmatrix} \hat{L}_3 & \\ & \hat{L}_6 \end{bmatrix}$$

where Lhat_3 is formed as in equation (4.7) and Lhat_6 is formed the same way, by replacing nodes 1 and 2 with nodes 4 and 5.

$$H = Q\hat{L}\left[\begin{array}{cccc} I & 0 & 0 & 0 \\ & \tilde{D}_1 & \tilde{U}_1 B_1 \tilde{U}_2^T & \tilde{U}_1 R_1 B_3 R_4^T \tilde{U}_4^T & \tilde{U}_1 R_1 B_3 R_5^T \tilde{U}_5^T \\ 0 & I & 0 & 0 \\ \tilde{U}_2 B_1^T \tilde{U}_1^T & & \tilde{U}_2 R_2 B_3 R_4^T \tilde{U}_4^T & \tilde{U}_2 R_2 B_3 R_5^T \tilde{U}_5^T \\ 0 & 0 & I & 0 \\ \tilde{U}_4 R_4 B_6 R_1^T \tilde{U}_1^T & \tilde{U}_4 R_4 B_6 R_2^T \tilde{U}_2^T & \tilde{D}_4 & \tilde{U}_4 B_4 \tilde{U}_5^T \\ 0 & 0 & 0 & I \\ \tilde{U}_5 R_5 B_6 R_1^T \tilde{U}_1^T & \tilde{U}_5 R_5 B_6 R_2^T \tilde{U}_2^T & \tilde{U}_5 B_4^T \tilde{U}_4^T & \tilde{D}_5 \end{array}\right]\hat{L}^T Q^T$$

$$P = \begin{bmatrix} P_1 & & & \\ & P_2 & & \\ & & P_3 & \\ & & & P_4 \end{bmatrix}$$

$$H = Q\hat{L}P\begin{bmatrix} \tilde{D}_1 & \tilde{U}_1 B_1 \tilde{U}_2^T & \tilde{U}_1 R_1 B_3 R_4^T \tilde{U}_4^T & \tilde{U}_1 R_1 B_3 R_5^T \tilde{U}_5^T \\ \tilde{U}_2 B_1^T \tilde{U}_1^T & \tilde{D}_2 & \tilde{U}_2 R_2 B_3 R_4^T \tilde{U}_4^T & \tilde{U}_2 R_2 B_3 R_5^T \tilde{U}_5^T \\ \tilde{U}_4 R_4 B_6 R_1^T \tilde{U}_1^T & \tilde{U}_4 R_4 B_6 R_2^T \tilde{U}_2^T & \tilde{D}_4 & \tilde{U}_4 B_4 \tilde{U}_5^T \\ \tilde{U}_5 R_5 B_6 R_1^T \tilde{U}_1^T & \tilde{U}_5 R_5 B_6 R_2^T \tilde{U}_2^T & \tilde{U}_5 B_4^T \tilde{U}_4^T & \tilde{D}_5 \end{bmatrix} P^T \hat{L}^T Q^T$$

by (4.7) and (4.8)

Note: The paper does not state what the P matrices look like, but that information is not necessary to check the correctness of the HSS Cholesky factors. All we need to know is that the P_j's are a set of orthogonal matrices that transform the above matrix into the sparser matrix in the previous step. The authors assume that these P's exist, so I will too.

Using the substitutions in equations (4.5) and (4.6), this expression becomes:

$$H = Q\hat{L}P\begin{bmatrix} D_3 & U_3 B_3 U_6^T \\ U_6 B_3^T U_3^T & D_6 \end{bmatrix} P^T \hat{L}^T Q^T$$

But this inner matrix is now in 2x2 HSS block form, so we can use equations (4.7) and (4.8) on it directly to get:

$$H = Q\hat{L}P\begin{bmatrix} Q_3 & \\ & Q_6 \end{bmatrix}\begin{bmatrix} \hat{L}_3 & \\ & \hat{L}_6 \end{bmatrix}\begin{bmatrix} P_3 & \\ & P_6 \end{bmatrix} D_7 \begin{bmatrix} P_3^T & \\ & P_6^T \end{bmatrix}\begin{bmatrix} \hat{L}_3^T & \\ & \hat{L}_6^T \end{bmatrix}\begin{bmatrix} Q_3^T & \\ & Q_6^T \end{bmatrix} P^T \hat{L}^T Q^T$$

where D_7 can be decomposed into $L_7 L_7^T$.

We have now written the original matrix H in terms of matrices that we already know from the generalized HSS Cholesky decomposition algorithm. We don't know what the P matrices are, but we know that they expand matrices in a particular way, so we can simulate their effect. I have used this set of equations in the cholHSS() function to check the correctness of the generalized Cholesky factors.