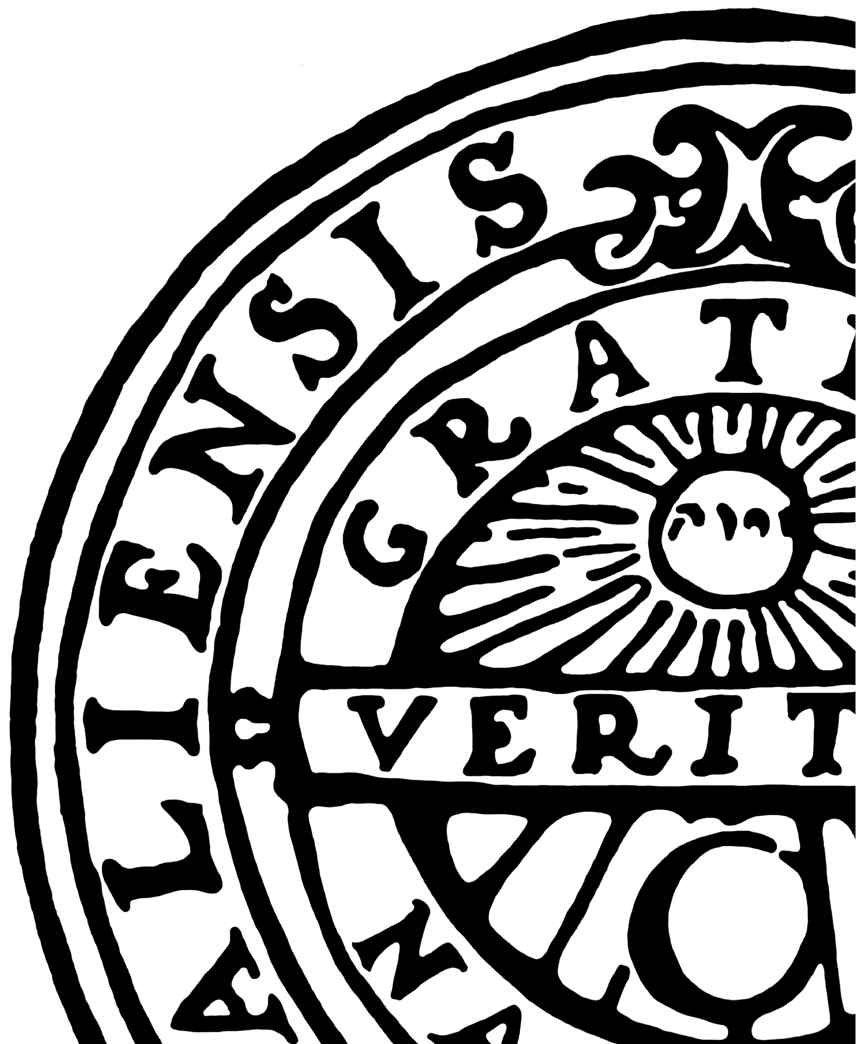
UPPSALA
UNIVERSITET

Efficient estimation of systematic radar errors

An implementation in C++ of the solution of a sparse linear system, using the CHOLMOD library for sparse matrices.

Frans Eliasson
Elisabeth Linnér

Report in Scientific Computing Advanced Course



Efficient Estimation of Systematic Radar Errors

An Implementation in C++ of the Solution of a Sparse Linear System, Using the CHOLMOD Library for Sparse Matrices

Frans Eliasson   Elisabeth Linnér

June 15, 2008

Abstract

The problem formulation for this project was provided by Saab Systems.

The task is to solve, as fast as possible, a sparse linear system of size 500 to 5000 unknowns, arising repeatedly in a radar tracking system. Further, it is required to extract the diagonal of the exact inverse of the coefficient matrix.

This report describes the undertaken solution approach and its program implementation in C++, based on the CHOLMOD[1] library. A brief analysis of the properties of the coefficient matrix, and suitable ways of approaching a problem of this kind, has been performed. A comparison with another program realization, presently used in practice, is also included.

Contents

1 Problem Formulation **5**

2 Background **5**
- 2.1 The Multi Sensor Tracker . . . 5
  - 2.1.1 Bias Compensation . . . 6
  - 2.1.2 The Matrix $\mathbf{A}$ . . . 6
  - 2.1.3 The Right Hand Side $\vec{b}$ . . . 7
  - 2.1.4 The Vector $\vec{M}$ . . . 7
  - 2.1.5 The Solution $\vec{x}$ . . . 7
  - 2.1.6 The Diagonal of $\mathbf{A}^{-1}$ . . . 7
- 2.2 Current Solution . . . 7

3 Alternative Solution Approaches **7**
- 3.1 Removal of the Manual Parameters . . . 7
- 3.2 Matrix Storage . . . 8
  - 3.2.1 Triplet Format . . . 8
  - 3.2.2 Compressed Row Storage (CRS) . . . 8
  - 3.2.3 Compressed Column Storage (CCS) . . . 9
- 3.3 Iterative vs. Direct Methods . . . 9
- 3.4 Permutation . . . 9
- 3.5 Cholesky Update . . . 11
- 3.6 Parallel Algorithms . . . 11
- 3.7 Programming Language . . . 12
- 3.8 Libraries . . . 12

4 Solution **13**
- 4.1 Algorithm . . . 13

5 Implementation **14**

6 Testing **15**
- 6.1 Testing architecture . . . 15
- 6.2 Accuracy . . . 15
- 6.3 Speedup . . . 15
- 6.4 Profiling . . . 15

7 Results **15**
- 7.1 Accuracy . . . 15
- 7.2 Speedup . . . 15

8 Discussion and Conclusions **16**
- 8.1 Why the Goal Was Only Partly Met . . . 16
- 8.2 Profiling . . . 17
- 8.3 Future Improvements . . . 17

Acknowledgements **18**

A The CHOLMOD Implementation **21**

B Makefile **31**

C The CHOLMOD Documentation **32**

D The Code for Simulating the Test Matrices **40**

E The Functions Used in the Current Implementation **42**

1 Problem Formulation

The task is to solve a linear system of equations of the form

$$\mathbf{A}\vec{x} = \vec{b} \tag{1.1}$$

where $\mathbf{A}$ is symmetric, positive definite. In addition, the diagonal of $\mathbf{A}^{-1}$ is to be computed. This should be done repeatedly, with an interval of about 60 seconds. Howeverm the total solution time must not be more than 30 seconds on the target architecture.

The number of unknowns varies between 500 and 5000. The matrix $\mathbf{A}$ has a sparse block structure, where each block is of size 16 by 16 elements. The diagonal elements of $\mathbf{A}$ are always non-zero. The density of the simulated test versions of the coefficient matrix $\mathbf{A}$, which correspond to typical real-life examples, is about 15%. The MATLAB code, which is used to generate the test matrices, can be found in Appendix D. The positioning of the non-zero elements in $\mathbf{A}$ can vary between calculations. When a change occurs, it is typically a change of a few hundred elements. However, for realistic examples, it must be assumed that all elements can change. An example of what the structure of $\mathbf{A}$ is shown in Figure 1.1.

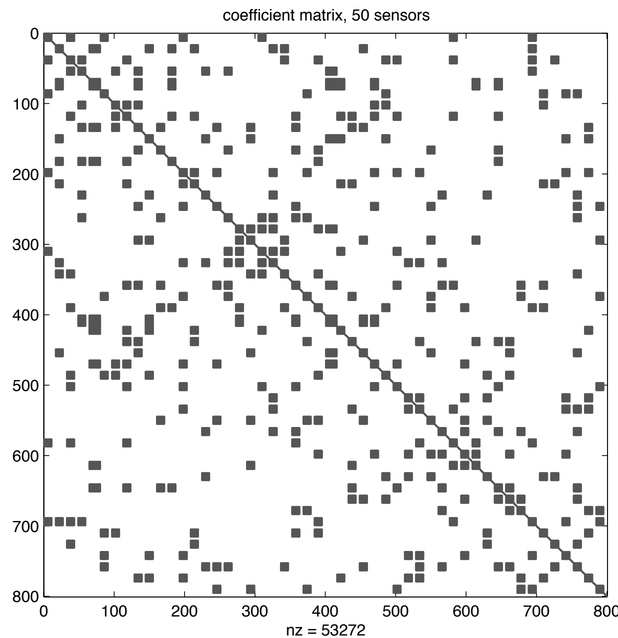


Figure 1.1: Example of the structure of the coefficient matrix.

It turns out that some unknowns of the system are redundant. The positions of these variables are given by a binary vector $\vec{M}$, in which the 1's mark that the corresponding rows and columns of $\mathbf{A}$, and entries in $\vec{x}$ and $\vec{b}$, can be eliminated from the system.

The problem should be solved as efficiently as possible using C, C++, Fortran or Ada.

2 Background

2.1 The Multi Sensor Tracker

To make radar surveillance more effective, several radars can be connected in a network. The data from the radars is assembled by a computer program, and a picture of the moving targets within the covered area is constructed. The product provided by Saab for this purpose is called MST, Multi Sensor Tracker, and works as follows.

Each radar of the network has a number of parameters defining, for example, its position and north alignment. For some radars, the values of these parameters can be rather inaccurate. For example, the position of a mobile radar can be incorrect by a couple of hundred meters. If two radars register the same target, and one of them has erroneous parameter values, the computer program will not be able to compute the motion of the target accurately. It may even interpret the information as two different targets, as can be seen in Figure 2.1.

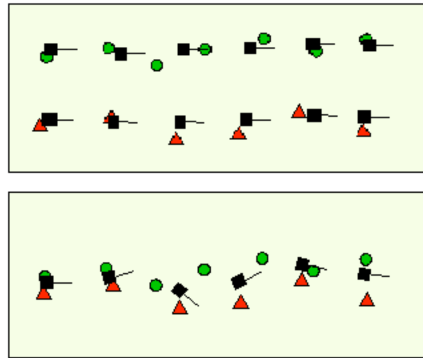


Figure 2.1: Examples of erroneous interpretations of target traces, arising from inaccurate parameter values.

2.1.1 Bias Compensation

To compensate for the inaccuracies, the values of 16 bias parameters are estimated once a minute. The standard deviations of these parameters are also calculated. Using this information, the computer program can make up for the systematic errors of the data, and a more precise approximation of the target movements can be done. The estimations are done using least squares approximation. The result of using bias compensation is showed in Figure 2.2.

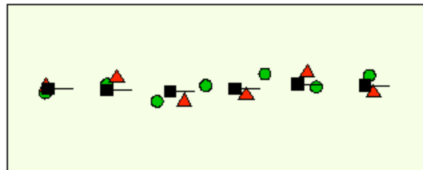


Figure 2.2: Using bias compensation, the target traces of Figure 2.1 can be considerably improved.

2.1.2 The Matrix A

The coefficient matrix of the system of equations is an information matrix of the radar network. Each radar is given an ID number. The 16 by 16 block $\mathbf{a}_{i,j}$ of the matrix $\mathbf{A}$ then corresponds to the relations between the parameters of radars with ID numbers i and j. $\mathbf{A}^{-1}$ is the covariance matrix of the bias parameters, and its diagonal elements are the variances the parameters. The inaccuracy of a parameter can always be compensated for in more than one way. For example, an erroneous position parameter of one radar can be remedied either by adjusting this position parameter, or by adjusting the position parameters of all the other radars. As an effect, $\mathbf{A}$ will always be singular, and the system of equations will not have a unique solution. However, it is probable that the estimations of most of the parameters are fairly correct. To favor the solutions that are close to zero, and make $\mathbf{A}$ positive definite, a diagonal matrix is added to it. $\mathbf{A}$ will still be ill-conditioned, and thus very sensitive to noise. Therefore, double precision is needed to store the elements of $\mathbf{A}$.

Type of radar	Parameters
2D	$1 - 6$
3D	$7 - 11$
SSR	$12 - 16$

Table 2.1: The bias parameters corresponding to the different types of antennas.

2.1.3 The Right Hand Side $\vec{b}$

The right hand side vector comes from the least squares approximation and has a more complex physical meaning, which does not need to be described here.

2.1.4 The Vector $\vec{M}$

Basically, there are two types of radars: 2D and 3D. 3D radars, naturally, use more parameters than 2D-radars. Some radars also have an antenna called SSR, which needs five additional parameters. The unused parameters are set manually, and there is no need for error approximation of those. The vector $\vec{M}$ contains information on which parameters are relevant.

2.1.5 The Solution $\vec{x}$

The vector $\vec{x}$, the solution of the system of equations, contains the expectation values of the bias parameters, which describe the systematic radar errors.

2.1.6 The Diagonal of $\mathbf{A}^{-1}$

The diagonal elements of $\mathbf{A}^{-1}$ are approximations of the variances for the corresponding values of $\vec{x}$.

2.2 Current Solution

In the current implementation of the MST, one can choose wether or not to take the sparsity of the coefficient matrix **A** into acount during the solution process. The choice is made at compile time. Using either option, the matrix **A** is stored as a dense matrix, but different sets of instructions are used.

3 Alternative Solution Approaches

As the implementation is required not to be bounded to a particular target computer architecture, all optimization must be done by improving the implementation of the solution, minimizing the number of arithmetic operations and the number of memory accesses. Both goals are achieved by choosing a good algorithm, and the second by choosing an efficient way of storage. Different alternatives are discussed below.

3.1 Removal of the Manual Parameters

For the parameters that are set manually, there is no need to compute the errors. They are always set to zero, since the values of these parameters can't be estimated. Using the information of the vector $\vec{M}$, containing markings for all the manual parameters, the corresponding rows and columns of **A** and $\vec{b}$ can be removed. This reduces **A** to about 40 % of its original size, and removes about 20 % of the non-zero elements. As a result, the matrix needs less space in memory, and the number of calculations is also reduced.

Conclusion

The information about manual parameters should be used to decrease the size of the matrix, thus removing unnecessary calculations, and decreasing the size of **A**.

3.2 Matrix Storage

By definition, the number of non-zero elements in a sparse matrix is $O(n)$, i.e. it is proportional to the side of the matrix (n). Comparing the total number of matrix elements ($O(n^2)$) to the number of non-zero elements ($O(n)$), it is seen that by storing only the non-zeros of the matrix, a lot of space can be saved. If all the data can be fitted into the cache, memory accesses are saved, and the program can run faster. There are various ways to store a sparse matrix efficiently.

3.2.1 Triplet Format

For each non-zero element, its value, row index and column index are stored. For a matrix with N non-zero elements, $3N$ entries are required. See Figure 3.1.

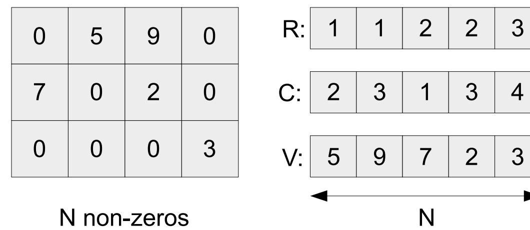


Figure 3.1: The triplet storage format.

3.2.2 Compressed Row Storage (CRS)

Three arrays are required to store the matrix. V contains the values of the non-zero elements of the matrix, stored row-wise. C holds the corresponding column indices to the entries of V. R contains row pointers; the ith row of the matrix begins at the R(i)th element of C or V. The last element of R contains the first index out of bounds of C and V. In this way, R($i+1$) - R(i) gives the number of non-zero elements in row i.

A matrix of size m by n with N non-zero elements requires $2N+m$ entries in memory. See Figure 3.2.

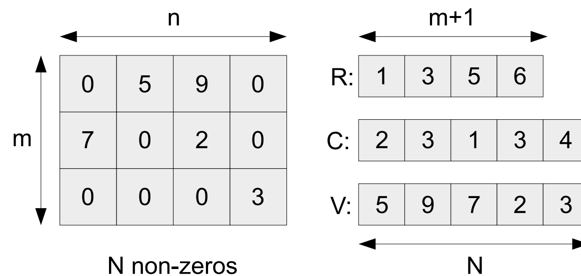


Figure 3.2: The compressed row storage format.

3.2.3 Compressed Column Storage (CCS)

This format is very similar to CRS, but the non-zero entries of the matrix are stored column-wise in V. R contains their row indices, and C contains the column pointers. A matrix of size m by n with N non-zero elements requires $2N + n$ entries in memory. See Figure 3.3.

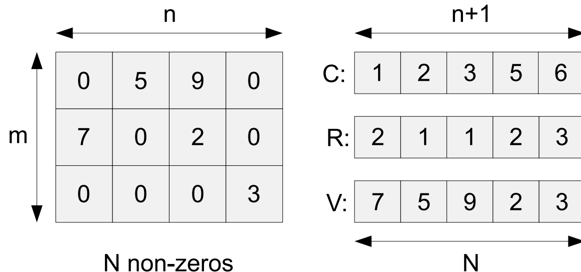


Figure 3.3: The compressed column storage format.

Conclusion

For the matrix **A**, CRS or CCS will save the most space in memory. Since **A** is symmetric, it doesn't matter which one is chosen; they will give identical results. Triplet format would have saved more space if N had been less than n and m, but this is not the case with the matrix **A**. The symmetry of **A** could also be used to minimize the amount of memory used, since only one half of the matrix, plus the diagonal, needs to be stored.

3.3 Iterative vs. Direct Methods

As was mentioned in the problem definition, the problem is to be solved for several time instances. The solutions of the sequence of systems are related to each other. If an iterative solution method is used, the solution from the previous time instance will make a very good starting guess for the next solution. However, as some recent comparisons between the performance of modern sparse direct solvers and preconditioned iterative solution methods show[12], the iterative solution methods become faster only for large enough systems, for example, for $n < 10^6$. Therefore, since the matrices of the target application are relatively small, no speedup would be gained from using an iterative solution method, and an advanced sparse direct solution method is a better choice.

Conclusion

As nothing would be gained from using an iterative method, the problem is solved directly. Cholesky factorization is used to simplify the solution of the linear system and the inversion of the matrix.

3.4 Permutation

The general way to solve a linear system of equations, like Equation 1.1, using a direct solution method, is to first factorize the matrix as a product of two triangular matrices,

$$\mathbf{A} = \mathbf{LU} \tag{3.1}$$

where **L** and **U** are lower and upper triangular, respectively. If **A** is symmetric, the so-called Cholesky factorization for symmetric matrices is used instead,

$$\mathbf{A} = \mathbf{LL}^T \tag{3.2}$$

where $\mathbf{L}$ is referred to as the Cholesky factor of $\mathbf{A}$. When the matrix has been factorized, the solution of 1.1 is computed in two steps, as follows:

$$\mathbf{L}\vec{y} = \vec{b} \tag{3.3}$$

$$\mathbf{L}^T\vec{x} = \vec{y} \tag{3.4}$$

For dense matrices, the cost of computing the Cholesky factor is $O(\frac{1}{3}n^3)$ (compared to a cost of $O(\frac{2}{3}n^3)$ for LU-factorization), and the cost of solving the two triangular systems of equations is $O(2n^2)$. However, the computational complexity of a sparse direct solver may vary, and depends heavily on the sparsity structure of the matrix $\mathbf{A}$. It is well known that if a sparse matrix with a general sparsity pattern is factorized, the resulting Cholesky factor can be nearly dense due to newly appeared non-zero entries during the factorization, referred to as "fill-in". Too much fill-in will substantially increase the arithmetic cost and memory requirements of the factorization itself, and of solving the triangular systems.

A general technique to limit fill-in is to first permute the matrix $\mathbf{A}$, in order to impose a proper non-zero structure. Among the best known non-zero patterns which produce minimal fill-in are, for example, block diagonal, banded and block bordered structures. In this project, four reordering strategies were compared in MATLAB, using test matrices simulated by `generateBlockMatrix.m` (Appendix D), to compare the reduction of the number of non-zeros in the Cholesky factor:

- Minimum Degree (`colmmd`, standard MATLAB function)
- Approximate Minimum Degree (`colamd`, standard MATLAB function)
- Reverse Cuthill McKee (`symrcm`, standard MATLAB function)
- Nested Dissection (`cs_nd` from the CSparse[3] library)

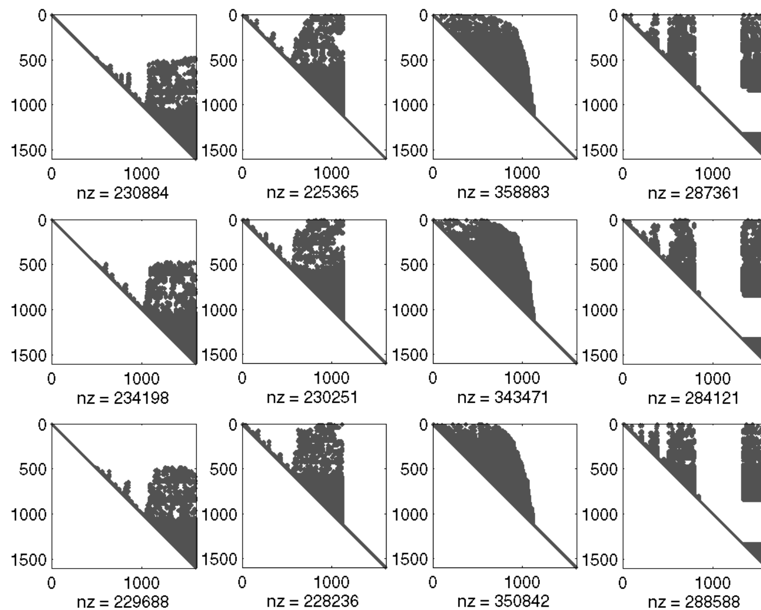


Figure 3.4: The columns represent the following permutations: `colmmd`, `colamd`, `symrcm` and `cs_nd`, respectively. The variable nz contains the number of non-zero entries in the resulting Cholesky factor. Each row corresponds to a test matrix.

Conclusion

As can be seen in Figure 3.4, for the target matrices, Approximate Minimum Degree (AMD)[13] reordering results in the least number of non-zero element in the $\mathbf{L}$.

3.5 Cholesky Update

Provided that the non-zero pattern of **A** does not change, and only som of the matrix entries are updated between two time instances, a new Cholesky factorization can be obtained using the Cholesky update method. This method typically has a calculation cost of $O(n^2)$, where n is the side of the coefficient matrix. A real-life recording done by the MST was provided by Saab Systems. From this recording, the number of elements that changes from zero to non-zero, for each calculation, were retrieved, see Figure 3.5.

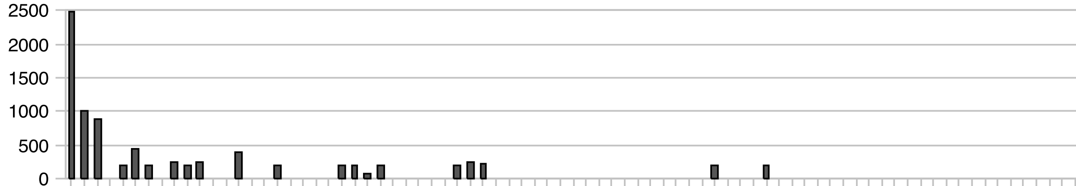


Figure 3.5: The number of elements that changes from zero to non-zero during each minute of a recording of the MST output. The recording was done while the MST was started, and the rapid changes of the first half are due to the system initialization. The second half of the diagram shows a better representation of the typical behavior of the system.

If the Cholesky update method is to be used in the implementation, the matrix must be checked for changes in its non-zero pattern, once for every time instance, before the problem is solved. For dense matrices, checking the matrix for changes and performing the Cholesky update cost $O(n^2)$ operations, compared to $O(\frac{1}{3}n^3)$ operations for using full Cholesky factorization every time. Assuming that the non-zero pattern changes at a ratio α of the time instances, the average cost to produce the Cholesky factorization for a dense matrix of the same size is

$$(1 + (1 - \alpha))n^2 + \frac{\alpha}{3}n^3 \tag{3.5}$$

The n^3-term is still present, since a full Cholesky factorization still needs to be performed if the non-zero pattern changes. Thus, for dense matrices, the cost will still be $O(n^3)$. For sparse matrices, the required number of arithmetic operations depend on the non-zero pattern of the matrix.

Conclusion

For dense matrices, using the Cholesky update method will result in speedup only if α is very small. The arithmetic costs of computing the Cholesky factorization and the Cholesky update for the sparse matrix **A** are not known. Because of this fact, and since the value of α cannot be estimated from the recording of the MST, Cholesky updating is left for future optimization.

3.6 Parallel Algorithms

As most hardware architectures today are multi-core, some investigation regarding parallel algorithms was done.

Due to the fact that the Cholesky factorization, as well as the solution of a linear system of equatins with a triangular coefficient matrix, are inherently serial, gaining parallelism is not a trivial task. There is recent research on this topic, from which it can be seen that the matrix structure plays an important role. Matrix inversion, however, implemented as n solutions of linear systems, with the already factorized matrix **A** as the coefficient matrix, implies perfect parallelism. Another possibility is to use Strassen's algorithm[16] and investigate its suitability in this context.

A special type of parallelization is the Single Instruction Multiple Data (SIMD)[14] capability in most modern x86 architectures. In theory, provided that BLAS uses this intrinsic CPU feature, the throughput can be doubled, by going from double precision (8 Byte) to float precision (4 Byte). Unfortunately, float precision yields incorrect solutions, as **A** is very poorly conditioned.

Some brief investigation into libraries for parallelization was done. Two widely used libraries are OpenMP[8] and MPI[9]. OpenMP supports gcc and msvc compilers via pragma directives.

Conclusion

Parallelization would probably bring considerable speedup of the computation of the matrix inverse. However, this could not be done within this project, due to time constraints.

3.7 Programming Language

Due to lack of time and experience, no other language than C/C++ was considered.

3.8 Libraries

For operations on dense matrices, the Basic Linear Algebra Subroutines (BLAS) is the de facto standard application programming interface. However, it is primally intended for dense matrices. Writing the needed algorithms for sparse matrices depending directly on the BLAS is unfeasible for this project due to complexity and time constraint. Therefore some set of libraries with necessary functionality were needed. There are many freeware programming libraries available for dealing with sparse matrices. A list of library candidates was compiled: CSparse[3], CHOLMOD[1], hypre[4], ILNumerics.Net[5], PETSc[6] and Trilinos[7]. Out of these, all were eliminated but two: CHOLMOD and PETSc. The cases against each of the other libraries were as follows:

- Trilinos does not have complete Cholesky factorization, but merely incomplete dito.
- hypre is GPL[10] licensed, which prohibits commercial use in closed source software12, which would cause problems for Saab.
- ILNumerics.Net was in a development stage, as of April 2008, and did not yet have any sparse methods implemented.
- A comparison of CSparse and CHOLMOD, using their MATLAB interfaces, proved CHOLMOD to superior to Csparse (see Figure 3.6). PETSc contained no MATLAB interface and could therefore not be part of the comparison.

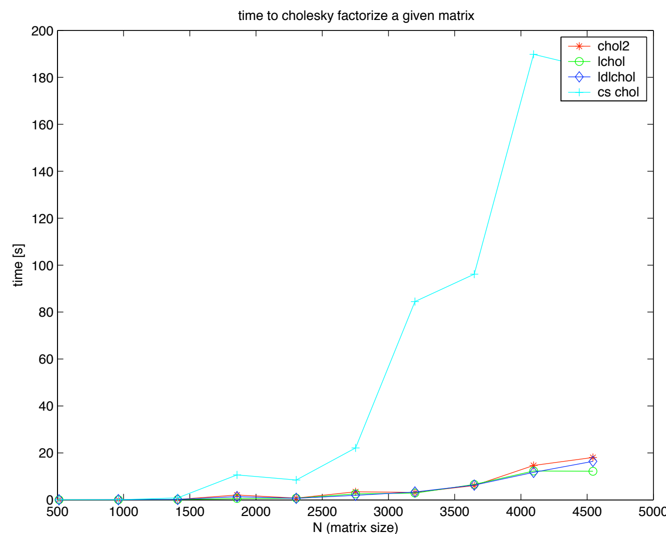


Figure 3.6:

Conclusion

Due to time constraints, PETSc could not be tested. Thus, CHOLMOD was chosen for the implementation. The necessary CHOLMOD functions are released under the GNU LGPL[11] license, which means that they can be used in closed source software.

4 Solution

4.1 Algorithm

The problem is solved as follows:

1. **Removal of redundant variables**
   The vector $\vec{M}$ is traversed, and when a non-zero element is encountered, the corresponding row and column of $\mathbf{A}$ are removed. So are the corresponding elements of $\vec{b}$.

2. **Reordering**
   $\mathbf{A}$ and b are permuted, using the AMD reordering algorithm[13]. Both sides of the system are multiplied by the transpose of the permutation matrix $\mathbf{P}$ from the left:
   $$\mathbf{P}^T\mathbf{A}\vec{x} = \mathbf{P}^T\vec{b} \tag{4.1}$$
   The solution vector $\vec{x}$ is substituted into
   $$\vec{x} = \mathbf{P}\vec{\hat{x}} \tag{4.2}$$
   The resulting system is
   $$\mathbf{P}^T\mathbf{A}\mathbf{P}\vec{\hat{x}} = \mathbf{P}^T\vec{b} \tag{4.3}$$
   which is rewritten into
   $$\hat{\mathbf{A}}\vec{\hat{x}} = \vec{\hat{b}} \tag{4.4}$$

3. **Cholesky factorization**
   $\hat{\mathbf{A}}$ is factorized intot
   $$\hat{\mathbf{A}} = \mathbf{L}\mathbf{L}^T \tag{4.5}$$

4. **Solving**
   The linear system of Equation 4.4 is solved in two steps:
   $$\mathbf{L}\vec{y} = \vec{\hat{b}} \tag{4.6}$$
   $$\mathbf{L}^T\vec{\hat{x}} = \vec{y} \tag{4.7}$$

5. **Inversion**
   $\hat{\mathbf{A}}$ is inverted. The inversion can be regarded as the solving of n systems of equations, where the solution vector of every system corresponds to a column of $\hat{\mathbf{A}}^{-1}$ and the right hand side is the corresponding column of the identity matrix $\mathbf{I}$:
   $$\hat{\mathbf{A}}\hat{\mathbf{A}}^{-1} = \mathbf{I} \tag{4.8}$$
   The diagonal $\vec{\hat{d}}$ of $\hat{\mathbf{A}}^{-1}$ is then extracted.

6. **Reordering**
   Inverse AMD permutation restores the elements of $\vec{\hat{x}}$ and $\vec{\hat{d}}$ to the original ordering. Since the permutation matrix $\mathbf{P}$ is orthogonal, the restoring process is simple:
   $$\vec{x} = \mathbf{P}\vec{\hat{x}} \tag{4.9}$$
   $$\vec{d} = \mathbf{P}\vec{\hat{d}}\mathbf{P}^T \tag{4.10}$$

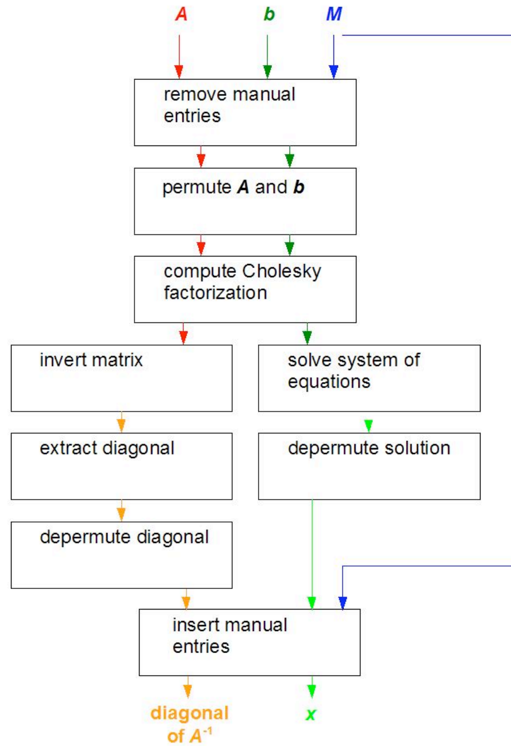


Figure 4.1: The flowchart shows the different stages of the algorithm.

7. **Restoring redundant variables**
   The solution vector $\vec{x}$ and the diagonal $\vec{d}$ are expanded to the original size For every non-zero element of $\vec{M}$, a zero element is inserted in both $\vec{x}$ and $\vec{d}$.

The algorithm is also depicted in Figure 4.1.

5 Implementation

The CHOLMOD library, which is used for this implementation, is written in C. It uses the CCS format to store the matrices. It also uses symmetry to reduce the required space when possible. The corresponding code can be found in Appendix A. The `cholmod_sparse` matrix struct, the `cholmod_factor` struct and the documentation of the CHOLMOD functions, that are used to solve this problem, can be found in Appendix C.

Step 1 is carried out before the matrix is converted from the original format into the `cholmod_sparse` format. At the same time, the number of non-zero elements, which is needed to allocate the `cholmod_sparse` variable, is computed. The conversions from the original format to the `cholmod_sparse` format are done by the functions `convertMatrix` and `convertVector`, written for this project.

Step 2 is performed by two different functions. The permutation vector is computed by `cholmod_analyze`, which analyzes the matrix $\mathbf{A}$. It is then stored in a `cholmod_factor` struct. The permutation of $\vec{b}$ is performed by the `cholmod_spsolve` function, which is a solver for sparse linear systems.

In **step 3**, $\hat{\mathbf{A}}$ is factorized by the `cholmod_factorize` function, and the factorization is stored in the `cholmod_factor` struct. $\mathbf{A}$ itself is never permuted, but the effects of the permutation are considered when forming the factor object.

Steps 4 and **5** are performed by `cholmod_spsolve`. As mentioned above, the inversion of $\hat{\mathbf{A}}$ is treated as a series of linear systems, one for every column of $\hat{\mathbf{A}}^{-1}$. The diagonal of $\hat{\mathbf{A}}^{-1}$ is extracted and stored in a vector by the function `invertDiagonal`.

Step 6 is also performed by `cholmod_spsolve`, as the permutation vector is stored in the `cholmod_factor` object.

In **step 7**, the manual parameters are put back into the diagonal and the solution to the equation system. At the same time, the vectors are converted to back to the original format. All this is done by the function `restoreVector`. The resulting vectors can then be returned to the program.

6 Testing

The code that was used to generate the test matrices can be found in Appendix D. A random vector was used as a right hand side for the system of equations.

6.1 Testing architecture

All tests were carried out on an UltraSPARC III, 1.3 GHz, with two cores. Each has an L1 data cache of 64 kB, an L1 instruction cache of 32 kB and an L2 cache of 1 MB.

6.2 Accuracy

For some test matrices of different sizes, the vector $\vec{x}$ and the diagonal of the inverse were computed by MATLAB and compared to the resulting vectors of the CHOLMOD implementation.

6.3 Speedup

Two solutions of the problem were compared: the current algorithms used in the MST, and the implementation using the CHOLMOD library. Identical test matrices, of sizes corresponding to 20 to 300 sensors, were given to the two programs. The average time of computation was measured for four matrices of each size. 300 sensors corresponds to 4800 unknowns, but larger matrices could not be tested, due to the memory constraints of MATLAB.

6.4 Profiling

The code was compiled with the `-p` flag and run through the `prof` profiler, which gave the output in Table 6.1. During profiling, a test matrix corresponding to 300 radar sensors was used, so that calculation time would be more important than the time taken to convert between the formats.

The `cholmod_amd` function is called during factorization by `cholmod_analyze` and `cholmod_factorize`. The function `cholmod_rowcolcounts` is also called by `cholmod_analyze`.

7 Results

7.1 Accuracy

Compared to the results calculated by MATLAB, both the vector $\vec{x}$ and the inverse of the diagonal are correct at least to the sixth significant digit. However, the elements of the diagonal are shuffled.

7.2 Speedup

Figure 7.1 shows a comparison between the currently used algorithms and the CHOLMOD algorithms. The CHOLMOD implementation is about 10 times faster than the current implementation. Figure 7.2 shows the average runtime for the CHOLMOD implementation only. Unfortunately,

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
78.6	104.88	104.88			cholmod_amd
13.2	17.66	122.54			cholmod_rowcolcounts
4.8	6.38	128.92			_fini
1.4	1.84	130.76	1	1840.	_mcount
0.9	1.17	131.93			cholmod_spsolve
0.5	0.65	132.58			cholmod_reallocate_column
0.4	0.59	133.17			cholmod_pack
0.0	0.06	133.23			cholmod_copy_dense
0.0	0.04	133.27			cholmod_aat
0.0	0.03	133.30			cholmod_transpose_sym
0.0	0.02	133.32			cholmod_etree
0.0	0.01	133.33			cholmod_analyze
0.0	0.01	133.34			cholmod_realloc_multiple
0.0	0.01	133.35			amd_2
0.0	0.01	133.36			cholmod_postorder
0.0	0.01	133.37			cholmod_free_dense
0.0	0.00	133.37	1	0.	main

Table 6.1: Profiling result

only problems corresponding to 200 sensors, or less, are solved within 30 seconds on the architecture that was used for the testing. However, that's more than twice as many as the current implementation can handle.

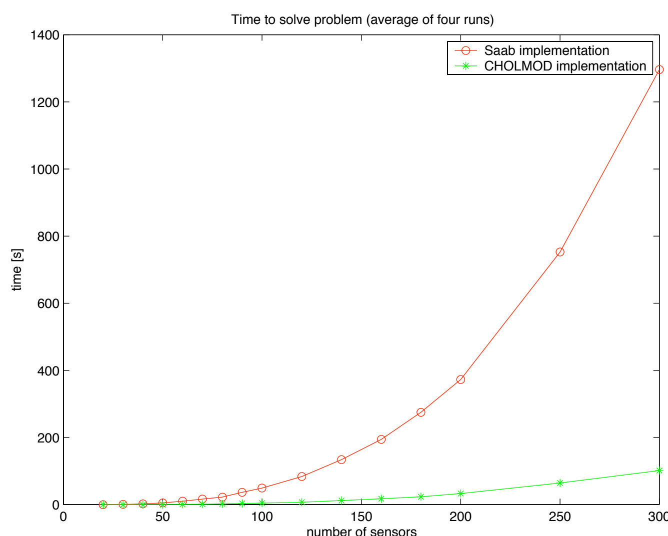


Figure 7.1: The runtimes of the CHOLMOD implementation, and the original MST implementation, for problems of different sizes.

8 Discussion and Conclusions

8.1 Why the Goal Was Only Partly Met

The UltraSPARC III is not a very fast processor. According Saab Systems' supervisor of the project, the code is likely to be run on a fairly new Intel processor, which will probably be faster. The conclusion is that nothing can be said until the code has been tested on the system on which it is supposed to be run.

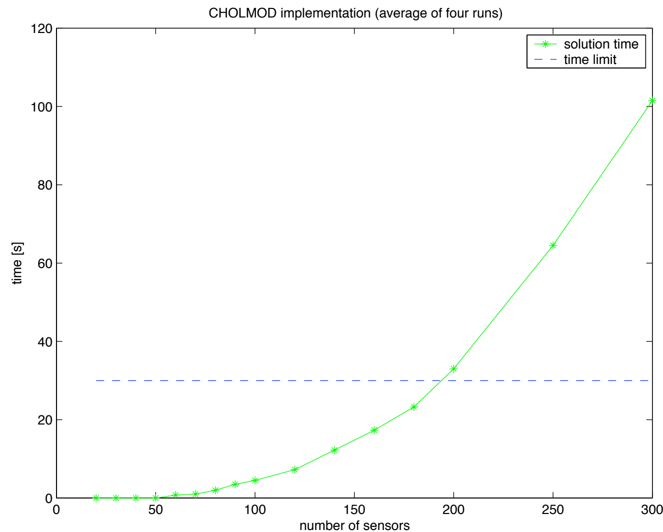


Figure 7.2: The runtime of CHOLMOD implementation for different problem sizes. The blue line marks the 30 seconds limit.

The incorrect ordering of the diagonal elements of $\mathbf{A}^{-1}$ might depend on misuse of the CHOLMOD functions, or on a bug in the library. Due to time constraints, this could not be investigated any further.

8.2 Profiling

The functions that take the most time, according to the profiling output, are cholmod_amd and cholmod_rowcolcount. Both are only called by other CHOLMOD functions. Thus, it seems that it would be rather pointless trying to optimize the parts of the code that were written for this project; the time taken to solve the problem depends on the implementation and usage of the CHOLMOD library.

8.3 Future Improvements

There is room for considerable improvements in the code appended to this report. The optimizations implemented were very limited due to time constraint. Optimizations not implemented include the following issues:

- The matrix inversion, which is the most computationally heavy part of the code, can easily be parallelized using either OpenMP or MPI. Either Strassen's algorithm[16] could be investigated, or the current inversion could be parallelized. Running the BLAS itself with multithreading enabled should be avoided since all implementations, as of May 2008, has a performance bug[2] that severely affects CHOLMOD performance.
- The Cholesky updating method can be used instead of full Cholesky factorization when the non-zero pattern of the coefficient matrix does not change.
- Provided that the target architecture is known, using specialized compilers and analyzing different optimizations can certainly speed up the computations.
- Since the size of $\mathbf{A}$ is constant during the runtime of the MST, memory should be allocated only once, saving time for both allocation and deallocation. This can easily be achieved, as this program will be converted to a function in the MST system. A pointer to $\mathbf{A}$ can be given to the function as an input. However, some parts of the CHOLMOD library would need to be modified, since many of its functions allocate memory internally.

- Other reordering strategies could be investigated, such as imposing block bordered structure.
- Using a block-version of the algorithm, the solution will be easier to parallelize. Block algorithms are also cache aware, since the block size can be adjusted to the architecture and size of the cache.
- The blocks of **A** could be treated as dense. The new, cache-aware BLAS, using the so-called packed storage data format[15], could then be investigated.
- The radars could be ordered so that most of the overlapping is concentrated to one part of the matrix. The matrix would then automatically have a block-diagonal structure, and would not need to be reordered. According to the profiler, the reordering function is the one that takes the most time, so a lot of speedup might be gained here.
- According to the profiling, it seems that calculating the permutation takes a lot of time. As long as the non-zero pattern of **A** does not change, the optimal permutation will be the same. Thus, the permutation matrix does not need to be calculated for every time instance. When changes occur, they are relatively small, and thus the optimal permutation will probably be similar to the one of the previous time instance. This means that it might be less computationally expensive to keep using the old permutation matrix, and have a little more fill-in, than computing a new permutation matrix every time the structure changes.
- The Cholesky factorization could be parallelized[17][18][19][20].

Acknowledgements

We would like to thank Lars Lindhagen, our supervisor from Saab Systems, for answering all our technical questions about the MST, and for his enthusiasm for the project and confindence in our work.

We would also like to thank Maya Neytcheva, our supervisor from Uppsala University, Department of Information Technology, for always patiently answeing our questions, no matter what time of day or what day of the week.

Morover, thanks to Timothy A. Davis for the information on sparse matrices and the CHOLMOD library.

Finally, Elisabeth Linnér would like to thank Niklas Fors for technical and moral support.

References

[1] *CHOLMOD*
Timothy A. Davis
University of Florida - Department of Computer and Information Science and Engineering
`http://www.cise.ufl.edu/~davis/`
November 2007

[2] *BLAS Performance Bug*
University of Florida - Department of Computer and Information Science and Engineering
`http://www.cise.ufl.edu/research/sparse/cholmod/blasbug.html`

[3] *Direct Methods for Sparse Linear Systems, and the CSparse Library*
Timothy A. Davis
University of Florida - Department of Computer and Information Science and Engineering
`http://www.cise.ufl.edu/research/sparse/CSparse/`
September 2006

[4] *Scalable Linear Solvers Project*
`https://computation.llnl.gov/casc/hypre/software.html`
December 15, 2006

[5] *ILNumerics.Net - Numeric Computing for .NET*
http://ilnumerics.net/
April 4, 2008

[6] *PETSc. the Portable, Extensible Toolkit for Scientific Computation*
http://www-unix.mcs.anl.gov/petsc/petsc-as/
May 23, 2007

[7] *The Trilinos Project*
http://trilinos.sandia.gov/
January 31, 2008

[8] *The OpenMP API Specification for Parallel Programming*
http://openmp.org/wp

[9] *The Message Passing Interface (MPI) Standard*
http://www-unix.mcs.anl.gov/mpi/

[10] *GNU General Public License*
http://www.gnu.org/licenses/gpl.html
June 29, 2007

[11] *GNU Lesser General Public License*
http://www.gnu.org/licenses/lgpl.html
June 29, 2007

[12] *A Comparison Between Two Solution Techniques to Solve the Equations of Glacial Rebound for an Elastic Earth*
Erik Bängtsson
Uppsala University - Departement of Information Technology
Björn Lund
Uppsala University - Departement of Earth Sciences
2006

[13] *An Approximate Minimum Degree Ordering Algorithm*
Patrick Amestoy
Timothy A. Davis
Iain S. Duff
SIAM Journal on Matrix Analysis and Applications
Volume 17, Number 4, Pages 886-905
1996

[14] *SIMD Architectures*
Jon Stokes
Ars Technica - The Art of Technology
March 21, 2000

[15] *A Recursive Formulation of Cholesky Factorization of a Matrix in Packed Storage*
Bjarne Stig Andersen
Danish Computing Center for Research and Education
Jerzy Wasniewski
Danish Computing Center for Research and Education
Fred G. Gustavson
IBM T. J. Watson Research Center, Yorktown Heights
ACM Transactions on Mathematical Software (TOMS)
Volume 27, Issue 2, Pages 214-244
2001

[16] *A Strassen-Newton Algorithm for High-Speed Parallelizable Matrix Inversion*
RNR-88-005

David H. Bailey
NASA Ames Research Center
Helaman R. P. Ferguson
Supercomputer Research Center
1988

[17] *A Parallel Cholesky Algorithm for the Solution of Symmetric Linear Systems*
R. R. Khazal
M. M. Chawla
International Journal of Mathematics and Mathematical Sciences
Volume 2004, Issue 25, Pages 1315-132
2004

[18] *Performance of Parallel Cholesky Factorization Algorithms Using BLAS*
G. R. Luecke
Heon Yun Jae
P. W. Smith
Journal of Supercomputing
Volume 6, Pages 315-329
1992

[19] *Sequential Performance Versus Scalability: Optimizing Parallel LU-decomposition*
Jens Simon
Jens-Michael Wierum
High Performance Computing and Networking
Volume 1067, Pages 627-632
1996

[20] *Parallel LU Factorization of Sparse Matrices on FPGA-based Configurable Computing Engines*
Xiofang Wang
Sotirios G. Ziavras
Concurrency and Computation: Practice and Experience Archive
Volume 16, Issue 4, Pages 319-343
2004

A The CHOLMOD Implementation

Listing 1: The code for solving the problem of Equation 1.1 using the CHOLMOD library.

```
#include "../MATRIX.HPP"
extern "C" {
#include "amd.h"
#include "ccolamd.h"
#include "cholmod.h"
#include "camd.h"
#include "colamd.h"
#include "UFconfig.h"
}
#include <fstream>
#include <iostream>
#include <string>
#include <ctime>
#include <cmath>

using namespace std;

//////////////////////////////////////////////////////////////////////////////
/// Converts a matrix of SAAB's matrix format into the cholmod_sparse format.
/// The original matrix is left undisturbed.
///
/// @param matrix     Input matrix.
/// @param nNonzeros  Number of non-zero elements.
/// @param common     Cholmod environment variable.
/// @return           Output matrix.
//////////////////////////////////////////////////////////////////////////////
cholmod_sparse *convertMatrix (
  const Matrix <double> *matrix,
  int                    nNonzeros,
  cholmod_common        *common
);
//////////////////////////////////////////////////////////////////////////////
/// Converts a vector of SAAB's matrix format into the cholmod_sparse format.
/// The original vector is left undisturbed.
///
/// @param vector     Input vector.
/// @param nNonzeros  Number of non-zero elements.
/// @param common     Cholmod environment variable.
/// @return           Output vector.
//////////////////////////////////////////////////////////////////////////////
cholmod_sparse *convertVector (
  const Vector <double> *vector,
  int                    nNonzeros,
  cholmod_common        *common
);
//////////////////////////////////////////////////////////////////////////////
/// Converts a vector of the cholmod_sparse format into SAAB's vector format.
/// The input vector is left undisturbed.
///
/// @param sparse     Input vector.
/// @param manual     Vector containing markings for manual entries.
/// @return           Output vector.
//////////////////////////////////////////////////////////////////////////////
Vector <double> *restoreVector (
  const cholmod_sparse  *sparse,
  const Vector <double> *manual
);
//////////////////////////////////////////////////////////////////////////////
/// Opens an ascii file and puts the data in a matrix of SAAB's matrix format.
///
/// @param filename   File to get matrix from.
/// @return           The matrix.
//////////////////////////////////////////////////////////////////////////////
Matrix <double> *readMatrix (
  const string filename
);
```

```
////////////////////////////////////////////////////////////////////////////
/// Opens an ascii file and puts the data in a column vector of SAAB's
/// vector format.
///
/// @param filename  File to get vector from.
/// @return          The vector.
////////////////////////////////////////////////////////////////////////////
Vector <double> *readVector (
  const string filename
);
////////////////////////////////////////////////////////////////////////////
/// Erases the manual entries of a matrix of SAAB's matrix class, and counts
/// the non-zero entries. The output matrix contains only the rows and columns
/// corresponding to non-manual parameters.
/// The variable nz is needed by the convertMatrix function.
///
/// @param nz        Outputs the number of non-zero elements.
/// @param original  The original matrix.
/// @param manual    Vector containing markings for manual entries.
/// @param newSize   Size of the output matrix.
/// @return          The matrix without the manual entries.
////////////////////////////////////////////////////////////////////////////
Matrix <double> *prepareMatrix (
  int                       &nz,
  const Matrix <double> *original,
  const Vector <double>        *manual,
  const int                  newSize
);
////////////////////////////////////////////////////////////////////////////
/// Erases the manual entries of a vector of SAAB's vector class, and counts
/// the non-zero entries. The output vector contains only the rows
/// corresponding to non-manual parameters.
/// The variable nz is needed by the convertVector function.
///
/// @param nz        Outputs the number of non-zero elements.
/// @param original  The original vector.
/// @param manual    Vector containting markings for manual entries.
/// @param newSize   Size of the output vector.
/// @return          The vector without the manual entries.
////////////////////////////////////////////////////////////////////////////
Vector <double> *prepareVector (
  int                       &nz,
  const Vector <double> *original,
  const Vector <double> *manual,
  const int                  newSize
);
////////////////////////////////////////////////////////////////////////////
/// Search for the value "key" in "array", starting the search at array[begin]
/// and searching until array[end-1].
////////////////////////////////////////////////////////////////////////////
int findElement(const int *array, int begin, int end, int key);
////////////////////////////////////////////////////////////////////////////
/// Sums the elements of a vector.
///
/// @param vector    Vector to be summed.
/// @return          Sum of the elements of the vector.
////////////////////////////////////////////////////////////////////////////
double sum (
  Vector <double> *vector
);
////////////////////////////////////////////////////////////////////////////
/// Inverts a cholmod_sparse matrix and returns its diagonal as a column vector.
///
/// @param L         Factor corresponding to the matrix to be inverted
/// @param common    Cholmond environment variable.
/// @return          Diagonal of the matrix inverse.
////////////////////////////////////////////////////////////////////////////
cholmod_sparse *invertDiagonal (
  cholmod_factor *L,
  cholmod_common *common
```

```
);


int main(int argc, char *argv[]) {

  cholmod_common  common[0]; // Maybe faster than common = new common.
  cholmod_sparse *A, *RHS, *temp1, *temp2, *temp3, *estimation,
                 *inverseDiagonal;
  cholmod_factor *L_factor;
  int             nnzMatrix, nnzRHS;
  string          filenameMatrix = "A_matrix.txt",
                  filenameManual = "A_manual.txt",
                  filenameRHS = "A_RHS.txt";
  clock_t         T1, T2, tConvert1, tConvert2, tConvert3, tCalculate;

  Vector <double> *saabManual = readVector (filenameManual);
  int notManual = saabManual->size() - static_cast <int> (sum (saabManual));
  Matrix <double> *saabMatrix = readMatrix (filenameMatrix);
  Vector <double> *saabRHS = readVector (filenameRHS);

  cout << "Starting timer...\n";
  T1 = clock();

  cholmod_start (common);
  common->final_ll = 1; // Choose Cholesky factorization on LL' form
  common->nmethods = 1; // AMD is the best permutation for this problem

  tConvert1 = clock();

  //
  // Conversion to the CHOLMOD format
  // --------------------------------
  Matrix <double> *saabNoManMatrix = prepareMatrix (
                                       nnzMatrix, saabMatrix,
                                       saabManual, notManual);
  Vector <double> *saabNoManRHS = prepareVector (
                                    nnzRHS, saabRHS, saabManual, notManual);
  A = convertMatrix (saabNoManMatrix, nnzMatrix, common);
  RHS = convertVector (saabNoManRHS, nnzRHS, common);

  tConvert2 = clock();

  //
  // Computation of the permutation vector and factorization of A
  // ------------------------------------------------------------
  L_factor = cholmod_allocate_factor (A->ncol, common);
  L_factor = cholmod_analyze (A, common);
  cholmod_change_factor (CHOLMOD_REAL, 1, 0, 1, 1, L_factor, common);
  cholmod_factorize (A, L_factor, common);

  //
  // Solution of the system of equations
  // -----------------------------------
  estimation = cholmod_allocate_sparse (
                 RHS->nrow, 1,
                 nnzRHS,
                 1, 1, 0, CHOLMOD_REAL,
                 common);
  temp1 = cholmod_spsolve(7, L_factor, RHS, common);
  temp2 = cholmod_spsolve(4, L_factor, temp1, common);
  estimation = cholmod_spsolve (5, L_factor, temp2, common);
  estimation->nz = new int[1];
  static_cast <int*> (estimation->nz)[0] = estimation->nzmax;
  temp3 = estimation;
  estimation = cholmod_spsolve(8, L_factor, temp3, common);

  //
  // Extraction of the diagonal of A^-1
  // ----------------------------------
  inverseDiagonal = invertDiagonal (L_factor, common);
```

```
  tCalculate = clock();

  //
  // Conversion to the original format
  // ------------------------------------
  Vector <double> *b = restoreVector (estimation, saabManual);
  Vector <double> *saabInvDiagonal = restoreVector (inverseDiagonal,
                                                    saabManual);

  tConvert3 = clock();

  cholmod_free_sparse (&A, common);
  cholmod_free_sparse (&RHS, common);
  cholmod_free_sparse (&temp1, common);
  cholmod_free_sparse (&temp2, common);
  cholmod_free_sparse (&temp3, common);
  cholmod_free_sparse (&estimation, common);
  cholmod_free_sparse (&inverseDiagonal, common);
  cholmod_free_factor (&L_factor, common);
  delete saabMatrix, saabNoManMatrix, saabManual, saabRHS, saabNoManRHS;

  cholmod_finish (common);
  T2 = clock();

  cout << "\n------ TIME ----------------------------------------------------\n\n";
  cout << "    Total time: " << (T2-T1)/CLOCKS_PER_SEC << " seconds." << endl;
  cout << "\n    Conversion time: " <<
        (tConvert2-tConvert1+tConvert3-tCalculate)/CLOCKS_PER_SEC
        << " seconds." << endl;
  cout << "    Calculation time: " << (tCalculate-tConvert2)/CLOCKS_PER_SEC
      << " seconds." << endl;
  cout << "\n\n--------------------------------------------------------------\n";
  cout << "\n    CLOCKS_PER_SEC: " << CLOCKS_PER_SEC << "\n\n";
  return 0;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * CONVERT MATRIX TO CHOLMOD FORMAT
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline cholmod_sparse *convertMatrix (
  const Matrix <double> *matrix,
  int                    nNonzeros,
  cholmod_common         *common
) {

  // storing in variables is faster than calling function all the time
  size_t nRows    = static_cast <size_t> (matrix->rows());
  size_t nColumns = static_cast <size_t> (matrix->cols());

  //
  // allocating return matrix
  // ------------------------
  cholmod_sparse *sparse = cholmod_allocate_sparse (
    nRows, nColumns,
    nNonzeros,
    1, 1, 1,
    CHOLMOD_REAL,
    common
  );
  sparse->nz = new int [nColumns];

  //
  // Copying the values
  // ------------------
  int n = 0; // counter; there won't be a new entry for every loop iteration
  double temp;
```

```
  for (int i = 0; i < nColumns; ++i) { // traversing columns
    static_cast <int*> (sparse->p ) [i] = n;
    static_cast <int*> (sparse->nz) [i] = 0;
    for (int j = 0; j < nRows; ++j) { // traversing rows
      temp = (*matrix)(i+1,j+1); // reversed indices is faster (symmetric)
      if (temp) {                // storing in variable saves function call
        static_cast <double*> (sparse->x) [n] = temp;
        static_cast <int*> (sparse->i) [n] = j;
        static_cast <int*> (sparse->nz) [i]++;
        n++;
      }
    }
  }
  static_cast <int*> (sparse->p) [nColumns] = n;

  return sparse;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * CONVERT VECTOR TO CHOLMOD FORMAT
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline cholmod_sparse *convertVector (
  const Vector <double> *vector,
  int                    nNonzeros,
  cholmod_common        *common
) {

  // storing in a variable is faster than calling function all the time
  size_t nRows = static_cast <size_t> (vector->size());

  //
  // Allocating result vector
  // ------------------------
  cholmod_sparse *sparse = cholmod_allocate_sparse (
                             nRows, 1,
                             nNonzeros,
                             1, 1, 0,
                             CHOLMOD_REAL,
                             common);
  static_cast <int*> (sparse->p) [0] = 0;
  sparse->nz = new int [1];
  static_cast <int*> (sparse->nz) [0] = 0;

  //
  // Copying the values
  // ------------------
  int n = 0; // counter; there won't be a new entry for every loop iteration
  double temp;
  for (int i = 0; i < nRows; i++) { // traversing rows
    temp = (*vector)(i+1); // storing in a variable saves one function call
    if (temp) {
      static_cast <double*> (sparse->x) [n] = temp;
      static_cast <int*> (sparse->i) [n] = i;
      static_cast <int*> (sparse->nz) [0]++;
      n++;
    }
  }
  static_cast <int*> (sparse->p) [1] = n;

  return sparse;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * RESTORE VECTOR TO ORIGINAL FORMAT
 *
```

```
 * Converts a vector of the CHOLMOD format bach to Saab's format, and puts
 * the manual parameters back.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline Vector <double> *restoreVector (
  const cholmod_sparse  *sparse,
  const Vector <double> *manual
) {

  // faster to store in a variable than calling a function all the time
  int length = manual->size();

  //
  // Allocating result vector
  // ------------------------
  Vector<double> *vector = new Vector<double>(length);

  //
  // Copying the values and inserting manual entries
  // -----------------------------------------------
  int n = 0; // counter; keeps track of the next element to copy
  for (int i = 1; i <= length; i++) {
    if (!(*manual)(i)) {
      (*vector)(i) = static_cast <double*> (sparse->x)[n];
      n++;
    }
    else  // For manual parameters, set to zero
      (*vector)(i) = 0.0;
  }

  return vector;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * READ MATRIX FROM FILE
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline Matrix <double> *readMatrix (const string filename) {

  fstream file (filename.c_str());
  //cout << (file.is_open() ? "File successfully open" : "File read error!")
  //     << endl;

  //
  // Read size of matrix
  // -------------------
  size_t size;
  file >> size;

  //
  // Allocating result matrix
  // ------------------------
  Matrix <double> *matrix = new Matrix <double> (size, size);

  //
  // Reading and copying values from file
  // ------------------------------------
  double temp;
  for (int i = 1; i <= size; ++i) {
    for (int j = 1; j <= size; ++j) {
      file >> temp;
      (*matrix) (i,j) = temp; // it didn't work to read directly into matrix
    }
  }

  file.close();
  return matrix;
}
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * READ VECTOR FROM FILE
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline Vector <double> *readVector (const string filename) {

  fstream file (filename.c_str());
  //cout << (file.is_open() ? "File successfully open" : "File read error!")
  //      << endl;

  //
  // Read length of vector
  // --------------------
  size_t length;
  file >> length;

  //
  // Allocating result vector
  // ------------------------
  Vector <double> *vector = new Vector <double> (length);

  //
  // Reading and copying values from file
  // ------------------------------------
  double temp;
  for (int i = 1; i <= length; ++i) {
    file >> temp;
    (*vector) (i) = temp; // it didn't work to read directly into the vector
  }

  file.close();
  return vector;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * REMOVE MANUAL ENTRIES AND COUNT NON-ZERO ELEMENTS OF MATRIX
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline Matrix<double> *prepareMatrix (
  int &nz,
  const Matrix<double> * original,
  const Vector<double> * manual,
  const int newSize
) {

  //
  // Allocating result matrix; original is not to be destroyed
  // ---------------------------------------------------------
  Matrix<double> *newMatrix = new Matrix<double>(newSize, newSize);

  //
  // Copying relevant elements and erasing manual entries
  // ----------------------------------------------------
  nz = 0;  // initialize non-zero counter
  int m = 1, n; // new row and column indices
  int size = manual->size(); // faster than calling function all the time
  double temp; // storing in variable saves one function call
  for (int i = 1; i <= size; i++) {
    n = 1; // reset column index
    if (!(*manual)(i)) {
      for (int j = 1; j <= size; j++) {
        if (!(*manual)(j)) {
          temp = (*original)(i,j);
          (*newMatrix)(m,n) = temp;
          n++;
```

```
            if (temp)
              nz++; // manual non-zero entries are not counted
          }
        }
        m++;
      }
    }

    return newMatrix;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * REMOVE MANUAL ENTRIES AND COUNT NON-ZERO ELEMENTS OF VECTOR
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline Vector<double> *prepareVector (
    int &nz,
    const Vector<double> *original,
    const Vector<double> *manual,
    const int newSize
) {

    //
    // Allocating result vector; original is not to be destroyed
    // ----------------------------------------------------------
    Vector<double> *newVector = new Vector<double>(newSize);

    //
    // Copying relevant elements and erasing manual entries
    // ----------------------------------------------------
    nz = 0; // initialize non-zero counter
    int n = 1; // new row index
    int size = manual->size(); // faster than calling function all the time
    double temp; // storing in variable saves one function call
    for (int i = 1; i <= size; i++) {
      if (!(*manual)(i)) {
        temp = (*original)(i);
        (*newVector)(n) = temp;
        if (temp)
          nz++; // manual non-zero entries are not counted
        n++;
      }
    }

    return newVector;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * SUM THE ELEMENTS OF A VECTOR
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
inline double sum(Vector<double> *vector) {

    double summa = 0;

    for (int i = 1; i <= vector->size(); i++) {
      summa += (*vector)(i);
    }

    return summa;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * SEARCHES FOR AN ELEMENT IN AN ARRAY
 *
```

```
 * Searches "array" for the element "key". Starts searching at the index
 * "begin" and stops at the index before "end".
 * Uses the Quick Search algorithm.
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
inline int findElement(const int *array, int begin, int end, int key) {

  end--;
  // first check point
  int test = begin + (end-begin) / 2, lastTest = -1;

  //
  // Searching the array
  // --------------------
  while (test != lastTest) { // Trying same index twice means no match found
    lastTest = test;
    if (array[test] > key) {
      end = test;
      test = begin + (end-begin) / 2;
    }
    else if (array[test] < key) {
      begin = test;
      test = begin + (end-begin+1) / 2; // Won't check last index without "+1"
    }
    else
      return test; // if found, return index where found
  }

  // if not found, return -1
  return -1;
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * INVERT MATRIX AND EXTRACT THE DIAGONAL OF THE INVERSE
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

inline cholmod_sparse *invertDiagonal(
                          cholmod_factor *L,
                          cholmod_common *common) {

  //
  // Allocating result matrix
  // ------------------------
  cholmod_sparse *diagonal = cholmod_allocate_sparse (
                                 L->n, 1, L->n,
                                 1, 1, 0,
                                 CHOLMOD_REAL, common);
  static_cast <int*> (diagonal->p)[0] = 0;
  static_cast <int*> (diagonal->p)[1] = 1;
  diagonal->nz = new int[1];
  static_cast <int*> (diagonal->nz)[0] = 0;

  //
  // Making unit matrix
  // ------------------
  cholmod_sparse *eye = cholmod_speye(L->n, L->n, CHOLMOD_REAL, common);

  //
  // Inverting matrix (result is put in eye)
  // ---------------------------------------
  cholmod_sparse *temp = cholmod_spsolve(4, L, eye, common);
  cholmod_free_sparse(&eye, common);
  eye = cholmod_spsolve(5, L, temp, common);
  // Using a temp variable wouldn't save any time, because it too would have
  // to be deallocated before return.

  //
  // Copying diagonal elements
  // -------------------------
```

```
  int n, nz = 0;
  for (int i = 0; i < L->n; i++) {
    n = findElement(static_cast <int*> (eye->i),
                    static_cast <int*> (eye->p)[i],
                    static_cast <int*> (eye->p)[i+1], i);
    if (n != -1){ //non-zero diagonal entry
      static_cast <double*>(diagonal->x)[nz]=static_cast <double*>(eye->x)[n];
      static_cast <int*> (diagonal->i)[nz] = i;
      static_cast <int*> (diagonal->nz)[0]++;
      nz++;
    }
  }

  cholmod_reallocate_sparse(nz, diagonal, common);
  cholmod_free_sparse(&eye, common);
  cholmod_free_sparse(&temp, common);
  return diagonal;
}
```

B Makefile

Listing 2: The Makefile that was used for compiling the code implementing CHOLMOD on the UltraSPARC III processor.

```
CC = g++

CCFLAGS =

SOURCES = main.cpp

# Include paths (with compiler flags):
AMDH       = -I../../AMD/Include
COLAMDH    = -I../../COLAMD/Include
UFCONFIGH  = -I../../UFconfig
CCOLAMDH   = -I../../CCOLAMD/Include
CHOLMODH   = -I../../CHOLMOD/Include
CAMDH      = -I../../CAMD/Include
METIS      = -I../../metis-4.0/Lib

INCLUDES = $(AMDH) $(COLAMDH) $(UFCONFIGH) $(CCOLAMDH) $(CHOLMODH) $(CAMDH) $(METIS)

# Library paths (with compiler flags)
CHOLMOD_PATH   = -L../../CHOLMOD/Lib/
AMD_PATH       = -L../../AMD/Lib/
COLAMD_PATH    = -L../../COLAMD/Lib/
CAMD_PATH      = -L../../CAMD/Lib/
CCOLAMD_PATH   = -L../../CCOLAMD/Lib/
UFCONFIG_PATH  = -L../../UFconfig/xerbla/
METIS_PATH     = -L../../metis-4.0
SUNPERF_PATH   = -L/opt/SUNWspro/lib/v8plusb # UltraSPARC III

CHOLMOD  = -lcholmod
AMD      = -lamd
COLAMD   = -lcolamd
CAMD     = -lcamd
CCOLAMD  = -lccolamd
UFCONFIG = -lcerbla
METIS    = -lmetis
SUNPERF  = -lsunperf

LIBRARIES = \
$(CHOLMOD_PATH) $(CHOLMOD) \
$(AMD_PATH) $(AMD) \
$(COLAMD_PATH) $(COLAMD) \
$(CAMD_PATH) $(CAMD) \
$(CCOLAMD_PATH) $(CCOLAMD) \
$(UFCONFIG_PATH) $(UFCONFIG) \
$(METIS_PATH) $(METIS) \
$(SUNPERF_PATH) $(SUNPERF)

all:
	$(CC) -O3 $(CCFLAGS) main.cpp $(FLAGS) $(INCLUDES) $(LIBRARIES) -o solver

debug:
	$(CC) -g $(CCFLAGS) main.cpp $(FLAGS) $(INCLUDES) $(LIBRARIES) -o solver

profile:
	$(CC) -p $(CCFLAGS) main.cpp $(FLAGS) $(INCLUDES) $(LIBRARIES)
	a.out
	prof

clean:
	rm -f *.out
```

C The CHOLMOD Documentation

```
/* ========================================================================== */
/* === Core/cholmod_sparse ================================================== */
/* ========================================================================== */

/* A sparse matrix stored in compressed-column form. */

typedef struct cholmod_sparse_struct
{
    size_t nrow ; /* the matrix is nrow-by-ncol */
    size_t ncol ;
    size_t nzmax ; /* maximum number of entries in the matrix */

    /* pointers to int or UF_long: */
    void *p ; /* p [0..ncol], the column pointers */
    void *i ; /* i [0..nzmax-1], the row indices */

    /* for unpacked matrices only: */
    void *nz ; /* nz [0..ncol-1], the # of nonzeros in each col.  In
 * packed form, the nonzero pattern of column j is in
* A->i [A->p [j] ... A->p [j+1]-1].  In unpacked form, column j is in
* A->i [A->p [j] ... A->p [j]+A->nz[j]-1] instead.  In both cases, the
* numerical values (if present) are in the corresponding locations in
* the array x (or z if A->xtype is CHOLMOD_ZOMPLEX). */

    /* pointers to double or float: */
    void *x ; /* size nzmax or 2*nzmax, if present */
    void *z ; /* size nzmax, if present */

    int stype ; /* Describes what parts of the matrix are considered:
 *
* 0:  matrix is "unsymmetric": use both upper and lower triangular parts
*     (the matrix may actually be symmetric in pattern and value, but
*     both parts are explicitly stored and used).  May be square or
*     rectangular.
* >0: matrix is square and symmetric, use upper triangular part.
*     Entries in the lower triangular part are ignored.
* <0: matrix is square and symmetric, use lower triangular part.
*     Entries in the upper triangular part are ignored.
*
* Note that stype>0 and stype<0 are different for cholmod_sparse and
* cholmod_triplet.  See the cholmod_triplet data structure for more
* details.
*/

    int itype ; /* CHOLMOD_INT:     p, i, and nz are int.
 * CHOLMOD_INTLONG: p is UF_long, i and nz are int.
 * CHOLMOD_LONG:    p, i, and nz are UF_long.  */

    int xtype ; /* pattern, real, complex, or zomplex */
    int dtype ; /* x and z are double or float */
    int sorted ; /* TRUE if columns are sorted, FALSE otherwise */
    int packed ; /* TRUE if packed (nz ignored), FALSE if unpacked
 * (nz is required) */
```

```
} cholmod_sparse ;
/* A symbolic and numeric factorization, either simplicial or supernodal.
 * In all cases, the row indices in the columns of L are kept sorted. */

typedef struct cholmod_factor_struct
{
    /* ------------------------------------------------------------------ */
    /* for both simplicial and supernodal factorizations */
    /* ------------------------------------------------------------------ */

    size_t n ; /* L is n-by-n */

    size_t minor ; /* If the factorization failed, L->minor is the column
 * at which it failed (in the range 0 to n-1).  A value
 * of n means the factorization was successful or
 * the matrix has not yet been factorized. */

    /* ------------------------------------------------------------------ */
    /* symbolic ordering and analysis */
    /* ------------------------------------------------------------------ */

    void *Perm ; /* size n, permutation used */
    void *ColCount ; /* size n, column counts for simplicial L */

    /* ------------------------------------------------------------------ */
    /* simplicial factorization */
    /* ------------------------------------------------------------------ */

    size_t nzmax ; /* size of i and x */

    void *p ; /* p [0..ncol], the column pointers */
    void *i ; /* i [0..nzmax-1], the row indices */
    void *x ; /* x [0..nzmax-1], the numerical values */
    void *z ;
    void *nz ; /* nz [0..ncol-1], the # of nonzeros in each column.
 * i [p [j] ... p [j]+nz[j]-1] contains the row indices,
 * and the numerical values are in the same locatins
 * in x. The value of i [p [k]] is always k. */

    void *next ; /* size ncol+2. next [j] is the next column in i/x */
    void *prev ; /* size ncol+2. prev [j] is the prior column in i/x.
 * head of the list is ncol+1, and the tail is ncol. */

    /* ------------------------------------------------------------------ */
    /* supernodal factorization */
    /* ------------------------------------------------------------------ */

    /* Note that L->x is shared with the simplicial data structure.  L->x has
     * size L->nzmax for a simplicial factor, and size L->xsize for a supernodal
     * factor. */

    size_t nsuper ; /* number of supernodes */
    size_t ssize ; /* size of s, integer part of supernodes */
    size_t xsize ; /* size of x, real part of supernodes */
    size_t maxcsize ; /* size of largest update matrix */
    size_t maxesize ; /* max # of rows in supernodes, excl. triangular part */
```

```
    void *super ; /* size nsuper+1, first col in each supernode */
    void *pi ; /* size nsuper+1, pointers to integer patterns */
    void *px ; /* size nsuper+1, pointers to real parts */
    void *s ; /* size ssize, integer part of supernodes */

    /* ---------------------------------------------------------------------- */
    /* factorization type */
    /* ---------------------------------------------------------------------- */

    int ordering ; /* ordering method used */

    int is_ll ; /* TRUE if LL', FALSE if LDL' */
    int is_super ; /* TRUE if supernodal, FALSE if simplicial */
    int is_monotonic ; /* TRUE if columns of L appear in order 0..n-1.
 * Only applicable to simplicial numeric types. */

    /* There are 8 types of factor objects that cholmod_factor can represent
     * (only 6 are used):
     *
     * Numeric types (xtype is not CHOLMOD_PATTERN)
     * -------------------------------------------
     *
     * simplicial LDL':  (is_ll FALSE, is_super FALSE).  Stored in compressed
     *     column form, using the simplicial components above (nzmax, p, i,
     *     x, z, nz, next, and prev).  The unit diagonal of L is not stored,
     *     and D is stored in its place.  There are no supernodes.
     *
     * simplicial LL': (is_ll TRUE, is_super FALSE).  Uses the same storage
     *     scheme as the simplicial LDL', except that D does not appear.
     *     The first entry of each column of L is the diagonal entry of
     *     that column of L.
     *
     * supernodal LDL': (is_ll FALSE, is_super TRUE).  Not used.
     *     FUTURE WORK:  add support for supernodal LDL'
     *
     * supernodal LL': (is_ll TRUE, is_super TRUE).  A supernodal factor,
     *     using the supernodal components described above (nsuper, ssize,
     *     xsize, maxcsize, maxesize, super, pi, px, s, x, and z).
     *
     *
     * Symbolic types (xtype is CHOLMOD_PATTERN)
     * -----------------------------------------
     *
     * simplicial LDL': (is_ll FALSE, is_super FALSE).  Nothing is present
     *     except Perm and ColCount.
     *
     * simplicial LL': (is_ll TRUE, is_super FALSE).  Identical to the
     *     simplicial LDL', except for the is_ll flag.
     *
     * supernodal LDL': (is_ll FALSE, is_super TRUE).  Not used.
     *     FUTURE WORK:  add support for supernodal LDL'
     *
     * supernodal LL': (is_ll TRUE, is_super TRUE).  A supernodal symbolic
     *     factorization.  The simplicial symbolic information is present
     *     (Perm and ColCount), as is all of the supernodal factorization
```

```
     *     except for the numerical values (x and z).
     */

    int itype ; /* The integer arrays are Perm, ColCount, p, i, nz,
 * next, prev, super, pi, px, and s.  If itype is
 * CHOLMOD_INT, all of these are int arrays.
 * CHOLMOD_INTLONG: p, pi, px are UF_long, others int.
 * CHOLMOD_LONG:    all integer arrays are UF_long. */
    int xtype ; /* pattern, real, complex, or zomplex */
    int dtype ; /* x and z double or float */

} cholmod_factor ;

/* ========================================================================== */
/* === cholmod_start ======================================================== */
/* ========================================================================== */

/* Initialize Common default parameters and statistics.  Sets workspace
 * pointers to NULL.
 *
 * This routine must be called just once, prior to calling any other CHOLMOD
 * routine.  Do not call this routine after any other CHOLMOD routine (except
 * cholmod_finish, to start a new CHOLMOD session), or a memory leak will
 * occur.
 *
 * workspace: none
 */

int CHOLMOD(start)
(
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_finish ======================================================= */
/* ========================================================================== */

/* The last call to CHOLMOD must be cholmod_finish.  You may call this routine
 * more than once, and can safely call any other CHOLMOD routine after calling
 * it (including cholmod_start).
 *
 * The statistics and parameter settings in Common are preserved.  The
 * workspace in Common is freed.  This routine is just another name for
 * cholmod_free_work.
 */

int CHOLMOD(finish)
(
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_allocate_sparse ============================================== */
/* ========================================================================== */

/* Allocate space for a matrix.  A->i and A->x are not initialized.  A->p
```

```
 * (and A->nz if A is not packed) are set to zero, so a matrix containing no
 * entries (all zero) is returned.  See also cholmod_spzeros.
 *
 * workspace: none
 */

cholmod_sparse *CHOLMOD(allocate_sparse)
(
    /* ---- input ---- */
    size_t nrow,/* # of rows of A */
    size_t ncol,/* # of columns of A */
    size_t nzmax,/* max # of nonzeros of A */
    int sorted,/* TRUE if columns of A sorted, FALSE otherwise */
    int packed,/* TRUE if A will be packed, FALSE otherwise */
    int stype,/* stype of A */
    int xtype,/* CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX */
    /* --------------- */
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_reallocate_sparse ============================================ */
/* ========================================================================== */

/* Change the size of A->i, A->x, and A->z, or allocate them if their current
 * size is zero.  A->x and A->z are not modified if A->xtype is CHOLMOD_PATTERN.
 * A->z is not modified unless A->xtype is CHOLMOD_ZOMPLEX.
 *
 * workspace: none
 */

int CHOLMOD(reallocate_sparse)
(
    /* ---- input ---- */
    size_t nznew,/* new # of entries in A */
    /* ---- in/out --- */
    cholmod_sparse *A,/* matrix to reallocate */
    /* --------------- */
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_free_sparse ================================================== */
/* ========================================================================== */

/* free a sparse matrix
 *
 * workspace: none
 */

int CHOLMOD(free_sparse)
(
    /* ---- in/out --- */
    cholmod_sparse **AHandle,/* matrix to deallocate, NULL on output */
    /* --------------- */
    cholmod_common *Common
```

```
)

/* ========================================================================== */
/* === cholmod_speye ======================================================== */
/* ========================================================================== */

/* Return a sparse identity matrix. */

cholmod_sparse *CHOLMOD(speye)
(
    /* ---- input ---- */
    size_t nrow,/* # of rows of A */
    size_t ncol,/* # of columns of A */
    int xtype,/* CHOLMOD_PATTERN, _REAL, _COMPLEX, or _ZOMPLEX */
    /* --------------- */
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_allocate_factor ============================================== */
/* ========================================================================== */

/* Allocate a simplicial symbolic factor, with L->Perm and L->ColCount allocated
 * and initialized to "empty" values (Perm [k] = k, and ColCount[k] = 1).
 * The integer and numerical parts of L are not allocated.  L->xtype is returned
 * as CHOLMOD_PATTERN and L->is_super are returned as FALSE.  L->is_ll is also
 * returned FALSE, but this may be modified when the matrix is factorized.
 *
 * This is sufficient (but far from ideal) for input to cholmod_factorize,
 * since the simplicial LL' or LDL' factorization (cholmod_rowfac) can
 * reallocate the columns of L as needed.  The primary purpose of this routine
 * is to allocate space for a symbolic factorization, for the "expert" user to
 * do his or her own symbolic analysis.  The typical user should use
 * cholmod_analyze instead of this routine.
 *
 * workspace: none
 */

cholmod_factor *CHOLMOD(allocate_factor)
(
    /* ---- input ---- */
    size_t n,/* L is n-by-n */
    /* --------------- */
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_free_factor ================================================== */
/* ========================================================================== */

/* Free a factor object.
 *
 * workspace: none
 */

int CHOLMOD(free_factor)
```

```
(
    /* ---- in/out --- */
    cholmod_factor **LHandle,/* factor to free, NULL on output */
    /* --------------- */
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_change_factor ================================================ */
/* ========================================================================== */

/* Convert a factor L.  Some conversions simply allocate uninitialized space
 * that meant to be filled later.
 *
 * If the conversion fails, the factor is left in its original form, with one
 * exception.  Converting a supernodal symbolic factor to a simplicial numeric
 * one (with L=D=I) may leave the factor in simplicial symbolic form.
 *
 * Memory allocated for each conversion is listed below.
 */

int CHOLMOD(change_factor)
(
    /* ---- input ---- */
    int to_xtype,/* convert to CHOLMOD_PATTERN, _REAL, _COMPLEX, or
 * _ZOMPLEX */
    int to_ll,/* TRUE: convert to LL', FALSE: LDL' */
    int to_super,/* TRUE: convert to supernodal, FALSE: simplicial */
    int to_packed,/* TRUE: pack simplicial columns, FALSE: do not pack */
    int to_monotonic,/* TRUE: put simplicial columns in order, FALSE: not */
    /* ---- in/out --- */
    cholmod_factor *L,/* factor to modify */
    /* --------------- */
    cholmod_common *Common
)

/* ========================================================================== */
/* === cholmod_analyze ====================================================== */
/* ========================================================================== */

/* Orders and analyzes A, AA', PAP', or PAA'P' and returns a symbolic factor
 * that can later be passed to cholmod_factorize. */

cholmod_factor *CHOLMOD(analyze)
(
    /* ---- input ---- */
    cholmod_sparse *A,/* matrix to order and analyze */
    /* --------------- */
    cholmod_common *Common
)
{
    return (CHOLMOD(analyze_p) (A, NULL, NULL, 0, Common)) ;
}

/* ========================================================================== */
/* === cholmod_factorize ==================================================== */
```

```
/* ========================================================================== */

/* Factorizes PAP' (or PAA'P' if A->stype is 0), using a factor obtained
 * from cholmod_analyze.  The analysis can be re-used simply by calling this
 * routine a second time with another matrix.  A must have the same nonzero
 * pattern as that passed to cholmod_analyze. */

int CHOLMOD(factorize)
(
    /* ---- input ---- */
    cholmod_sparse *A,/* matrix to factorize */
    /* ---- in/out --- */
    cholmod_factor *L,/* resulting factorization */
    /* --------------- */
    cholmod_common *Common
)
{
    double zero [2] ;
    zero [0] = 0 ;
    zero [1] = 0 ;
    return (CHOLMOD(factorize_p) (A, zero, NULL, 0, L, Common)) ;
}

/* ========================================================================== */
/* === cholmod_spolve ======================================================= */
/* ========================================================================== */

/* Given an LL' or LDL' factorization of A, solve one of the following systems:
 *
 * Ax=b     0: CHOLMOD_A also applies the permutation L->Perm
 * LDL'x=b     1: CHOLMOD_LDLt does not apply L->Perm
 * LDx=b     2: CHOLMOD_LD
 * DL'x=b     3: CHOLMOD_DLt
 * Lx=b     4: CHOLMOD_L
 * L'x=b     5: CHOLMOD_Lt
 * Dx=b     6: CHOLMOD_D
 * x=Pb     7: CHOLMOD_P apply a permutation (P is L->Perm)
 * x=P'b     8: CHOLMOD_Pt apply an inverse permutation
 *
 * where b and x are sparse.  If L and b are real, then x is real.  Otherwise,
 * x is complex or zomplex, depending on the Common->prefer_zomplex parameter.
 * All xtypes of x and b are supported (real, complex, and zomplex).
 */

cholmod_sparse *CHOLMOD(spsolve)     /* returns the sparse solution X */
(
    /* ---- input ---- */
    int sys,/* system to solve */
    cholmod_factor *L,/* factorization to use */
    cholmod_sparse *B,/* right-hand-side */
    /* --------------- */
    cholmod_common *Common
)
```

D The Code for Simulating the Test Matrices

Listing 3: The MATLAB code that was used to produce the test matrices.

```
function [a, manual] = generateBlockMatrix2(ns)

% Generate block-sparse matrix for TDB project course
% Manual vector added. Sent to students 080514
%clear

np = 16; % Number of parameters per sensors
nPsrPar = 11; % Number of PSR bias parameters per sensor
%ns = N;
nSsrRdr = floor(0.05 * ns); % Number of SSR radars
nOverlap = 4; % Number of sensors with overlapping coverage area (PSR)
nOverlapSsr = 0.1 * ns; % D:o (but for SSR radar)

N = np * ns; % Total number of bias parameters

manual = zeros(N, 1); % Manual vector. 1 = manual parameter

for i = 1 : ns
   rdr{i}.ssr = 0; % SSR radar?
   rdr{i}.overlap = nOverlap; % Number of overlapping sensors
end % i
for i = 1 : nSsrRdr
   ix = unidrnd(ns); % random number in the range [1,ns]
   rdr{ix}.ssr = 1;
   rdr{ix}.overlap = nOverlapSsr;
end % i
for i = 1 : ns
   if ~rdr{i}.ssr % if there's no SSR sensor
      manual((i-1)*np+nPsrPar+1 : i*np) = 1; % Sets the last 5 parameters
   end % if
end % i

% setting manual parameters
for i = 1:floor(ns*0.7)
     j = unidrnd(ns);
     manual((i-1)*np+7 : (i-1)*np+11) = 1;
end


overlap = zeros(ns); % Boolean matrix telling whether two sensors overlap

for i = 1 : ns
   overlap(i, i) = 1;
   for r = 1 : rdr{i}.overlap
      j = unidrnd(ns);
      overlap(i, j) = 1;
      overlap(j, i) = 1;
   end % r
end % i

a = zeros(N); % LHS matrix to form
for i = 1 : ns
   for j = 1 : ns
      if overlap(i, j)
         minI = (i - 1) * np + 1;
         maxI = i * np;
         minJ = (j - 1) * np + 1;
         maxJ = j * np;
         if rdr{i}.ssr;
            ii = np;
         else
            ii = nPsrPar;
         end % if
         if rdr{j}.ssr;
            jj = np;
         else
```

```
          jj = nPsrPar;
        end % if
        aBlock = zeros(np);
        aBlock(1:ii, 1:jj) = randn(ii, jj);
        a(minI:maxI, minJ:maxJ) = aBlock;
      end % if
    end % j
end % i
a = a + a'; % Make symmetric
a = a + diag(sum(abs(a)) + 1); % Make diagonal dominant
%spy(a)
%shg
```

E The Functions Used in the Current Implementation

Listing 4: The header of the Matrix¡Template¿ class.

```
#ifndef MATRIX_HPP
#define MATRIX_HPP

template <class T>  class Vector;

template <class T>
class Matrix
{
public:
    Matrix(int rows,int cols);
    Matrix(const Matrix<T>& M);
   virtual    ~Matrix();
   Matrix<T>& operator=(const Matrix<T>&);
   T&     operator()(int,int);
   const T&     operator()(int,int)     const;
   int          rows()                  const;
   int          cols()                  const;
   void        clear();                                  // Set all elements to 0
   Matrix<T>& operator += (const Matrix<T>&);
   Matrix<T>& operator -= (const Matrix<T>&);
   Matrix<T>& operator*=(T);
   Matrix<T>& operator/=(T);
   Matrix<T>  operator+(const Matrix<T>&)       const;
   Matrix<T>  operator-(const Matrix<T>&)       const;
   Matrix<T>  operator*(const Matrix<T>&)       const;
   Matrix<T>  operator/(T)                      const;
   Matrix<T>  operator*(T)                      const;
   Vector<T>  operator*(const Vector<T>&)       const;
   Matrix<T>  transpose()                       const;
protected:
   Matrix() {}
   T       *m_start,*m_end;              // start and end of data block
   T      **m_pMatrix,**m_matrix;        // pointer _matrix[1.._rows],p_matrix[0.._rows-1]
   int        m_size,m_rows,m_cols;       // Matrix size [_rows,_cols], _size=_rows*_cols
   T*       operator[](int)      const;
   void     initMatrix();
private:
   int        inRange(int,int)     const;
};

#include "Matrix.inl"

#endif
```

Listing 5: The functions of the Matrix¡Template¿ class.

```
#include "Vector.hpp"

template <class T>
inline    Matrix<T>::Matrix(int rows,int cols):m_rows(rows),m_cols(cols)
{
   if(rows<=0 || cols <=0) throw("Error in Matrix constructor");
   m_size=m_rows*m_cols;
   m_start = new T [m_size];             // Holds matrix elements
   m_pMatrix = new T* [m_rows];         // Pointers to each row
   initMatrix();
}

template <class T>
inline  Matrix<T>::Matrix(const Matrix<T>& M)  :
 m_rows(M.m_rows),m_cols(M.m_cols)
{
   m_size=M.m_size;
   m_start = new T [m_size];             // Holds matrix elements
   m_pMatrix = new T* [m_rows];         // Pointers to each row
   initMatrix();
```

```
    *this=M;
}

template <class T>
inline      Matrix<T>::~Matrix()
{
    if (m_start)
    {
        delete [] m_start;
        delete [] m_pMatrix;
    }
}

//  Protected members
template <class T>
inline     T*    Matrix<T>::operator[](int row)   const
{
    return m_matrix[row];
}

template <class T>
inline    void   Matrix<T>::initMatrix()
{
    m_end=m_start+m_size;
    m_matrix=m_pMatrix-1;                        // Fix for proper indexing [1...rows]
    m_matrix[1]=m_start-1;                       // Fix so that m_matrix[i]=row[1...cols]
    for(int i=1;i<m_rows;i++)
        m_matrix[i+1]=m_matrix[i]+m_cols;
}

// public members

template <class T>
inline   Matrix<T>& Matrix<T>::operator=(const Matrix<T>& M)
{
    if(this!=&M)
    {
        if(m_rows!=M.m_rows || m_cols!=M.m_cols)
            throw("Error in Matrix =");
         memcpy(m_start,M.m_start,m_size*sizeof(T));          // copy element
    }

     return *this;
}

template <class T>
inline    T&     Matrix<T>::operator()(int row,int col)
{
#ifndef NO_RANGE_CHECK
    if(!inRange(row,col))
        throw("index out of range in matrix");
#endif
    return m_matrix[row][col];
}

template <class T>
inline    const T&     Matrix<T>::operator()(int row,int col) const
{
#ifndef NO_RANGE_CHECK
    if(!inRange(row,col))
        throw("index out of range in matrix");
#endif
    return m_matrix[row][col];
}

template <class T>
inline   int    Matrix<T>::inRange(int row,int col)           const
{
    return   row>0 && row<=m_rows && col>0 && col<=m_cols;
}
```

```
template <class T>
inline    int    Matrix<T>::rows()       const
{
   return m_rows;
}

template <class T>
inline    int    Matrix<T>::cols()       const
{
   return m_cols;
}

template <class T>
inline  Matrix<T> Matrix<T>::transpose()    const
{
   int row,col;
   Matrix<T>   ret(m_cols,m_rows);
   for(row=1;row<=m_rows;row++)
      for(col=1;col<=m_cols;col++)
         ret[col][row]=m_matrix[row][col];
   return ret;
}

template <class T>
inline      void Matrix<T>::clear()
{
   memset(m_start,0,m_size*sizeof(T));
}

template <class T>
inline     Matrix<T>& Matrix<T>::operator += (const Matrix<T>& M)
{
   T   *p,*pM;
   if(m_rows!=M.m_rows || m_cols!=M.m_cols)
      throw("Error in Matrix +=");
   p=m_start;
   pM=M.m_start;                        // Set pointers
   while(p<m_end)
      *(p++)+=*(pM++);              // Add
   return *this;
}

template <class T>
inline      Matrix<T>& Matrix<T>::operator -= (const Matrix<T>& M)
{
   T   *p,*pM;
   if(m_rows!=M.m_rows || m_cols!=M.m_cols)
      throw("Error in Matrix +=");
   p=m_start;
   pM=M.m_start;                         // Set pointers
   while(p<m_end)
      *(p++)-=*(pM++);               // Subtract
   return *this;
}

template <class T>
inline      Matrix<T>& Matrix<T>::operator*=(T c)
{
   T   *p = m_start;
   while(p<m_end)
      *(p++) *= c;
   return *this;
}

template <class T>
inline  Matrix<T>& Matrix<T>::operator/=(T c)
{
   T   *p = m_start;
   while(p<m_end)
```

```
      *(p++) /= c;
   return *this;
}

template <class T>
inline  Matrix<T> Matrix<T>::operator+(const Matrix<T>& M)  const
{
   if(m_rows!=M.m_rows || m_cols!=M.m_cols)
      throw("Error in Matrix+Matrix");
   Matrix<T> ret(*this);
    return ret += M;
}

template <class T>
inline     Matrix<T> Matrix<T>::operator-(const Matrix<T>& M)  const
{
   if(m_rows!=M.m_rows || m_cols!=M.m_cols)
      throw("Error in Matrix-Matrix");
    Matrix<T> ret(*this);
    return ret -= M;
}

template <class T>
inline     Matrix<T>  Matrix<T>::operator*(const Matrix<T>& M)  const
{
   if(m_cols!=M.m_rows)
      throw("Error in Matrix*Matrix");
    Matrix<T> ret(m_rows,M.m_cols);      // Return matrix
   ret.clear();
    for(int row=1;row<=ret.m_rows;row++)          // Can be faster !!!
      for(int col=1;col<=ret.m_cols;col++)
         for(int i=1;i<=m_cols;i++)
            ret[row][col]+=m_matrix[row][i]*M.m_matrix[i][col];
   return ret;
}

template <class T>
inline    Matrix<T>  Matrix<T>::operator/(T s)               const
{
   Matrix<T> ret(*this);
   return    ret/=s;
}

template <class T>
inline    Matrix<T>  Matrix<T>::operator*(T s)               const
{
   Matrix<T> ret(*this);
   return ret*=s;
}

template <class T>
inline    Vector<T>  Matrix<T>::operator*(const Vector<T>& V) const
{
   if(V.size()!=m_cols)
      throw("Error in Matrix*Vector");
   Vector<T> ret(m_rows);
   ret.clear();
   for(int row=1;row<=m_rows;row++)
      for(int col=1;col<=m_cols;col++)
         ret(row) += m_matrix[row][col]*V(col);
   return ret;
}


template <class T>
inline Matrix<T>  operator*(T s,const Matrix<T>& p)
{
   return p*s;
}
```

```
template <class T>
inline Matrix<T>     Vector<T>::tensor(const Vector<T>& v)      const
{
   Matrix<T>   ret(size(),v.size());
   for(int r=1;r<=ret.rows();++r)
      for(int c=1;c<=ret.cols();++c)
         ret(r,c) = (*this)(r) * v(c);
   return ret;
}

template<class T>
inline Vector<T>  Vector<T>::operator*(const Matrix<T>& m)  const
{
   return m.Transpose()*(*this);
}

template<class T>
inline std::ostream& operator<<(std::ostream& os,const Matrix<T>& m)
{
   os<<"{{";
   for(int r=1;r<=m.rows();r++)
    {
      int c;
      for(c=1;c<m.cols();c++)
      {
         os<<m(r,c)<<',';
      }
      os<<m(r,c);
      if(r<m.rows())
      {
         os<<'}'<<std::endl<<'{';
      }
   }
   return os<<"}}"<<std::endl;
}
```

Listing 6: The header of the Vector¡Template¿ class.

```
#ifndef VECTOR_HPP
#define VECTOR_HPP


template <class T>  class Matrix;

/*******************************
*
*   This class represents a Vector. Special with numerical operators
*
*******************************/


// class used by sort algorithms
template<class T> class Comp
{
public:
  inline static bool less(T&a,T&b) {return a<b;}
};

template <class T>
class Vector : public Comp<T>
{
public:
    Vector(int len);
    Vector(const Vector<T>& V);
    virtual ~Vector();
   class Iter
   {
      friend class Vector<T>;
   public:
      Iter(T* c) : ce(c) {}      // Should nopt be public, needed for VCC
```

```
    Iter() {}
    operator T*()                {return  ce;}
    //    T*          operator->()      {return  ce;}
    T&        operator*()       {return *ce;}
    T&        operator[](int i) {return ce[i];}
    Iter  operator++()      {return Iter(++ce);}
    Iter  operator++(int)   {return Iter(ce++);}
    Iter  operator--()      {return Iter(--ce);}
    Iter  operator--(int)   {return Iter(ce--);}
    Iter  operator+(int i)  {return Iter(ce+i);}
    Iter  operator-(int i)  {return Iter(ce-i);}
    Iter& operator+=(int i) {ce+=i; return *this;}
    Iter& operator-=(int i) {ce-=i; return *this;}
    int          operator-(const Iter& ci)   {return ce - ci.ce;}
    int    operator==(const Iter i) const      {return ce == i.ce;}
    int    operator!=(const Iter i) const      {return ce != i.ce;}
    int    operator>(const Iter i) const      {return ce>i.ce;}
    int    operator<(const Iter i) const      {return ce<i.ce;}
    int    operator>=(const Iter i) const      {return ! (ce<i.ce);}
    int    operator<=(const Iter i) const      {return !(ce>i.ce);}
private:
    T  *ce;
};
class ConstIter
{
    friend class Vector<T>;
public:
    ConstIter(T* c) : ce(c) {}
    ConstIter() {}
    operator T const*()              {return ce;}
    //   T const*        operator->()      {return ce;}
    const T&       operator*()       {return *ce;}
    const T&       operator[](int i) {return ce[i];}
    ConstIter   operator++()      {return ConstIter(++ce);}
    ConstIter   operator++(int)   {return ConstIter(ce++);}
    ConstIter   operator--()      {return ConstIter(--ce);}
    ConstIter   operator--(int)   {return ConstIter(ce--);}
    ConstIter   operator+(int i)  {return ConstIter(ce+i);}
    ConstIter   operator-(int i)  {return ConstIter(ce-i);}
    ConstIter&   operator+=(int i) {ce+=i; return *this;}
    ConstIter&   operator-=(int i) {ce-=i; return *this;}
    int         operator-(const ConstIter& ci)        {return ce - ci.ce;}
    int         operator==(const ConstIter i)   const   {return ce == i.ce;}
    int         operator!=(const ConstIter i)   const   {return ce != i.ce;}
    int         operator>(const ConstIter i)   const   {return ce>i.ce;}
    int    operator<(const ConstIter i)         const   {return ce<i.ce;}
    int    operator>=(const ConstIter i)      const   {return ! (ce<i.ce);}
    int    operator<=(const ConstIter i)      const   {return !(ce>i.ce);}
private:
    T  *ce;
};

virtual void newSize(int len);
int   size()           const;
T&   operator()(int);
const T&   operator()(int) const;
const Vector<T>& operator=(const  Vector&);
Iter    begin()                {return Iter(m_first);}
Iter    end()                  {return Iter(m_last);}
Iter    rbegin()               {return Iter(m_last-1);}
Iter    rend()                 {return Iter(m_vector);}
ConstIter    begin()   const   {return ConstIter(m_first);}
ConstIter    end()     const   {return ConstIter(m_last);}
ConstIter    rbegin()  const   {return ConstIter(m_last-1);}
ConstIter    rend()    const   {return ConstIter(m_vector);}

void clear();
Vector<T>& operator *= (T);
Vector<T>& operator /= (T);
Vector<T>& operator += (const Vector<T>&);
```

```
    Vector<T>& operator -= (const Vector<T>&);
    T operator*(const Vector<T>&)              const;
    Vector<T> operator*(T)                     const;
    Vector<T> operator/(T)                     const;
    Vector<T> operator+(const Vector<T>&)      const;
    Vector<T> operator-(const Vector<T>&)      const;
    Vector<T> operator*(const Matrix<T>&)      const;
    Matrix<T> tensor(const Vector<T>&)         const;

    // routines to sort vector
    void heapSort(int n);
    void heapSort()    {heapSort(size());}
    void heapSortIndex(Vector<int>& index, int n);
    void heapSortIndex(Vector<int>& index)
        {heapSortIndex(index,size());}
    void qSort(int n);
    void qSort()    {qSort(size());}



protected:
    Vector()   {m_first = 0;};
    T        *m_first,*m_last,*m_vector;   // vector=first-1=vector[1...n]
    int        inRange(int) const;
};

#include "Vector.inl"

#endif
```

Listing 7: The functions of the Vector¡Template¿ class

```
#include <iostream>
#include <string.h>
#include <stdlib.h>

template <class T>
inline  Vector<T>::Vector(int len)
{
    if(len<0)
        len=1;
    m_first = new T[len];
    m_last=m_first+len;
    m_vector=m_first-1;
}

template <class T>
inline  Vector<T>::Vector(const Vector<T>& V)
{
    m_first = new T[V.size()];
    m_last=m_first+V.size();
    m_vector=m_first-1;
    memcpy(m_first,V.m_first,V.size()*sizeof(T));
}

template <class T>
inline    Vector<T>::~Vector()
{
    if(m_first != 0)
        delete[] m_first;
    m_first = 0;
}

template <class T>
inline  void    Vector<T>::newSize(int len)
{
    T *tmp;
    int old_len = m_last-m_first;
    int tmpLen;
    if(len<0)
```

```
        len=1;
    if(len == old_len)
        return;
    tmp=m_first;
    m_first = new T[len];
    m_last = m_first+len;
    memset(m_first, 0, len * sizeof(T));
    tmpLen =  (len<old_len)?len:old_len;
    memcpy(m_first,tmp, tmpLen*sizeof(T));
    delete [] tmp;
    m_vector=m_first-1;
}

template <class T>
inline    int  Vector<T>::size() const
{
    return m_last-m_first;
}

template <class T>
inline    T&   Vector<T>::operator()(int i)
{
#ifndef NO_RANGE_CHECK
    if(!inRange(i))
            throw("Index out of range in Vector");
#endif
    return m_vector[i];
}

template <class T>
inline const  T&   Vector<T>::operator()(int i) const
{
#ifndef NO_RANGE_CHECK
    if(!inRange(i))
            throw("Index out of range in Vector");
#endif
    return m_vector[i];
}

template <class T>
inline    int  Vector<T>::inRange(int i)  const
{
    return i>=1 && i<=m_last-m_first;
}

template <class T>
inline    const Vector<T>& Vector<T>::operator=(const  Vector<T>& v)
{
    if(this!=&v)
    {
        if(size()!=v.size())
            throw("Error in Vector = ");
        memcpy(m_first,v.m_first,v.size()*sizeof(T));
    }
    return *this;
}

template <class T>
inline      void Vector<T>::clear()
{
    memset(m_first,0,(m_last-m_first)*sizeof(T));
}

template <class T>
inline      Vector<T>& Vector<T>::operator *= (T s)
{
    T *p = m_first;
    while(p<m_last)
        *(p++) *= s;
    return *this;
```

```
}

template <class T>
inline      Vector<T>& Vector<T>::operator /= (T s)
{
   T *p = m_first;
   while(p<m_last) *(p++) /= s;
   return *this;
}

template <class T>
inline      Vector<T>& Vector<T>::operator += (const Vector<T>& v)
{
   T   *p = m_first,
       *pv = v.m_first;
   while(p<m_last)
       *(p++) += *(pv++);
   return *this;
}

template <class T>
inline      Vector<T>& Vector<T>::operator -= (const Vector<T>& v)
{
   T   *p=m_first,
       *pv=v.m_first;
   while(p<m_last) *(p++) -= *(pv++);
   return *this;
}

template <class T>
inline      T Vector<T>::operator*(const Vector<T>& v)  const               // scalar product
{
   T   *p   = m_first+1,
       *pv = v.m_first;
   T   retvalue = (*(p++))*(*(pv++));

   while(p<m_last)
       retvalue += (*(p++))*(*(pv++));
   return retvalue;
}

template <class T>
inline      Vector<T> Vector<T>::operator*(T s) const
{
   Vector<T> ret(*this);
   ret *= s;
   return ret;
}

template <class T>
inline      Vector<T> Vector<T>::operator/(T s) const
{
   Vector<T> ret(*this);
   ret /= s;
   return ret;
}

template <class T>
inline      Vector<T> Vector<T>::operator+(const Vector<T>& A) const
{
   Vector<T> ret(*this);
   ret += A;
   return ret;
}

template <class T>
inline      Vector<T> Vector<T>::operator-(const Vector<T>& A) const
{
   Vector<T> ret(*this);
   ret -= A;
```

```
    return ret;
}

template <class T>
inline std::ostream& operator<<(std::ostream& os,const Vector<T>& v)
{
    os<<'{';
    for(int i=1;i<v.size();i++)
        os<<v(i)<<',';
    return  os<<v(v.size())<<'}'<<endl;
}

template <class T>
inline Vector<T>  operator*(T s,const Vector<T>& p)
{
    return p*s;
}

// Sort functions
template<class T>  void  Vector<T>::heapSort(int n)
{
    int l,j,ir,i;
    T  rra;
    Iter vector = begin()-1;  // will be vector[1..n]

    if(n<=1)
        return;

    l=(n >> 1)+1;
    ir=n;
    for (;;)
    {
        if (l > 1)
            rra=vector[--l];
        else
        {
            rra=vector[ir];
            vector[ir]=vector[1];
            if (--ir == 1)
            {
                vector[1]=rra;
                return;
            }
        }
        i=l;
        j=l << 1;
        while (j <= ir)
        {
            if (j < ir && less(vector[j],vector[j+1]) )
                ++j;
            if (less(rra,vector[j]) )
            {
                vector[i]=vector[j];
                j += (i=j);
            }
            else
                j=ir+1;
        }
        vector[i]=rra;
    }
}

template<class T>  void  Vector<T>::heapSortIndex(Vector<int>& indexA,int n)
{
    int l,j,ir,indxt,i;
    T  q;
    Iter vector = begin()-1;  // will be vector[1..n]

    if(n<=1)
        return;
```

```
    Vector<int>::Iter index = indexA.begin();
    index -= 1;  // index[1..n]  will access array

    for (j=1;j<=n;j++) index[j]=j;
    l=(n >> 1) + 1;
    ir=n;
    for (;;)
    {
        if (l > 1)
            q=vector[(indxt=index[--l])];
        else
        {
            q=vector[(indxt=index[ir])];
            index[ir]=index[1];
            if (--ir == 1)
            {
                index[1]=indxt;
                return;
            }
        }
        i=l;
        j=l << 1;
        while (j <= ir)
        {
            if (j < ir && less(vector[index[j]],vector[index[j+1]]) )
                j++;
            if ( less(q,vector[index[j]]) )
            {
                index[i]=index[j];
                j += (i=j);
            }
            else j=ir+1;
        }
        index[i]=indxt;
    }
}

template<class T>  void  Vector<T>::qSort(int n)
{
    const int  M = 7;
    const int  NSTACK = 50;
    const int  FM = 7875;
    const int  FA = 211;
    const int  FC = 1663;
    int l=1,jstack=0,j,ir,iq,i;
    int istack[NSTACK+1];
    long int fx=0L;
    T  a;
    Iter vector = begin()-1;  // will be vector[1..n]
    if(n<=1)
        return;
    ir=n;
    for (;;)
    {
        if (ir-l < M)
        {
            for (j=l+1;j<=ir;j++)
            {
                a=vector[j];
                for (i=j-1;less(a,vector[i]) && i>0;i--)
                    vector[i+1]=vector[i];
                vector[i+1]=a;
            }
            if (jstack == 0)
                return;
            ir=istack[jstack--];
            l=istack[jstack--];
        }
        else
```

```
        {
            i=l;
            j=ir;
            fx=(fx*FA+FC) % FM;
            iq=l+((ir-l+1)*fx)/FM;
            a=vector[iq];
            vector[iq]=vector[l];
            for (;;)
            {
                while (j > 0 && less(a,vector[j]))
                    j--;
                if (j <= i)
                {
                    vector[i]=a;
                    break;
                }
                vector[i++]=vector[j];
                while (less(vector[i],a) && i <= n)
                    i++;
                if (j <= i)
                {
                    vector[(i=j)]=a;
                    break;
                }
                vector[j--]=vector[i];
            }
            if (ir-i >= i-l)
            {
                istack[++jstack]=i+1;
                istack[++jstack]=ir;
                ir=i-1;
            } else
            {
                istack[++jstack]=l;
                istack[++jstack]=i-1;
                l=i+1;
            }
            if (jstack > NSTACK)
            {
                cerr<<"NSTACK too small in QCKSRT"<<endl;
                exit (0);
            }
        }
    }
}
```

Listing 8: The header of the linear algebra functions.

```
#ifndef LIN_ALG_HPP
#define LIN_ALG_HPP

#include "Matrix.hpp"
#include "Vector.hpp"


// Full Cholesky stuff, but with a special, row-oriented matrix representation
namespace LinAlgFullCholesky
{
    // Full Cholesky decomposition stuff
    // Only the lower half of the matrix is stored, and may be addressed by the () operator!
    class FullCholesky
    {
    public:
        class MatrixNotPositiveDefinite {}; // Thrown by decompose() if matrix is not positive definite

        FullCholesky(int size);
        int      size()   const   {return m_matrix.size();} // Order of A

        // --> Methods that may be called before decomposition
        void    initiate(const Matrix<double>&); // Initiates to argument
```

```
      double    operator()(int i, int j)   const   {return (*m_matrix.m_rows(i))(j);} // For inspectin
      double&   operator()(int i, int j)           {return (*m_matrix.m_rows(i))(j);} // ...and modify

      // --> Overwriting Cholesky decomposition
      void    decompose();

      // --> Methods that may be called after decomposition (the original matrix is called A)
      void    backSub(Vector<double> &b)         const; // Solves A*x = b. The solution is stored in b
      void    inverseDiag(Vector<double> &res)       const; // Stores the diagonal of the inverse of A
      void    backSub_diag2lower(Matrix<double> &m)    const; // Solves A*x=m, where m is diagonal. Th
   private:
      // A matrix row
      class Row : public Vector<double>
      {
      public:
         Row(int size) : Vector<double>(size) {}
      };

      // Representation of lower half C of A or of Cholesky decomposition C of A (C * C^T = A)
      // Consists of several Row's of increasing length
      struct FCMatrix
      {
         FCMatrix(int size);
         int       size()                  const {return m_rows.size();}
         void    backSub(Vector<double> &b)    const; // Solves C*x = b. The solution is stored in b
         Vector<Row*>    m_rows;       // The rows of the lower half of the matrix to be decomposed
      };

      // Representation of transpose C^T of Cholesky decomposition C of A
      // Consists of several Row's of decreasing length
      struct FCTranspose
      {
         FCTranspose(int size);
         void    initiate(const FCMatrix &rhs); // Initiates us as transpose of 'rhs'
         int size()    const {return m_rows.size();}
         void    backSub(Vector<double> &b)    const; // Solves C^T*x = b. The solution is stored in b
         Vector<Row*>    m_rows;       // The rows of the lower half A^T
      };

      // Disable copy constructor and operator=
      FullCholesky(const FullCholesky&);
      const FullCholesky& operator=(const FullCholesky&);

      FCMatrix       m_matrix;            // The matrix A or its decomposition
      FCTranspose      m_transpose;          // The transpose of A. Initialized after decomposition.
      mutable Vector<double>   m_dummy;        // For temporary use in inverseDiag
   };
}

// Sparse Cholesky stuff
namespace LinAlgSparseCholesky
{
   // Sparse Cholesky decomposition stuff
   // Only the lower half of the matrix is stored, and may be addressed by the () operator!
   class SparseCholesky
   {
   public:
      class MatrixNotPositiveDefinite {}; // Thrown by decompose() if matrix is not positive definite

      SparseCholesky(int size, double tol);
      int       size()    const    {return m_matrix.size();} // Order of A

      // --> Methods that may be called before decomposition
      void    initiate(const Matrix<double>&); // Initiates to an unpacked matrix
      double    operator()(int i, int j)   const   {return (*m_matrix.m_rows(i))(j).m_val;} // For in
      double&   operator()(int i, int j)           {return (*m_matrix.m_rows(i))(j).m_val;} // ...and

      // --> Overwriting Cholesky decomposition
      void    decompose();
```

```
        // --> Methods that may be called after decomposition (the original matrix is called A)
        void    backSub(Vector<double> &b)        const; // Solves A*x = b. The solution is stored in b
        void    inverseDiag(Vector<double> &res)        const; // Stores the diagonal of the inverse of A
        void    backSub_diag2lower(Matrix<double> &m)    const; // Solves A*x=m, where m is diagonal. Th
    private:
        // A non-zero matrix entry
        struct Item
        {
            int       m_j;    // Column index. Row index is clear from its place in a row
            double    m_val;    // Value of entry
            void set(int j, double val) {m_j = j; m_val = val;}
        };

        // A matrix row. Only non-zero items are stored. The row is to be indexed by p, p1, pOld, ...
        class Row : public Vector<Item>
        {
        public:
            Row(int size) : Vector<Item>(size) {}
            void    reset()    {m_nItems = 0;} // Make us empty
            int m_nItems;    // OK to read item 1, ..., m_nItems
        };

        // Representation of lower half C of A or of Cholesky decomposition C of A (C * C^T = A)
        // Consists of several Row's of increasing length
        struct SCMatrix
        {
            SCMatrix(int size);
            int        size()                        const {return m_rows.size();}
            void    backSub(Vector<double> &b)    const; // Solves C*x = b. The solution is stored in b
            Vector<Row*>    m_rows;        // The rows of the lower half of the matrix to be decomposed
        };

        // Representation of transpose C^T of Cholesky decomposition C of A
        // Consists of several Row's of decreasing length
        struct SCTranspose
        {
            SCTranspose(int size);
            void    initiate(const SCMatrix &rhs); // Initiates us as transpose of 'rhs'
            int size()    const {return m_rows.size();}
            void    backSub(Vector<double> &b)    const; // Solves C^T*x = b. The solution is stored in b
            Vector<Row*>    m_rows;        // The rows of the lower half A^T
        };

        // Disable copy constructor and operator=
        SparseCholesky(const SparseCholesky&);
        const SparseCholesky& operator=(const SparseCholesky&);
        void    pack();                                                    // Full -> sparse conversion
        //void    unpack();                                                 // Sparse -> full conversion
        void    decompose_computeCii(const int i);                              // Computes diagonal eleme
        void    decompose_computeCij(const int i, const int j, const int pParam);   // Computes off-diag
        void    decompose_insertNewRow(int i);                                 // Inserts the new elements

        SCMatrix        m_matrix;            // The matrix A or its decomposition
        SCTranspose        m_transpose;          // The transpose of A. Initialized after decomposition.
        Row                m_newRow;             // For storing new items appearing during decomposition of
        const double    m_tol;               // Smaller values are treated as zero
        mutable Vector<double>    m_dummy;        // For temporary use in inverseDiag
        mutable Vector<int>       m_firstIx;        // For temporary use in inverseDiag
    };
}

#include "LinAlg.inl"

#endif
```

Listing 9: The linear algebra functions

```
#include <math.h>

// ------------------------------------------------------------------ FullCholesky
```

```
inline LinAlgFullCholesky::FullCholesky::FCMatrix::FCMatrix(int size)
: m_rows(size)
{
   for (int i = 1; i <= size; ++i)
   {
      m_rows(i) = new Row(i);
   }
}

inline LinAlgFullCholesky::FullCholesky::FCMatrix::~FCMatrix()
{
   for (int i = 1; i <= m_rows.size(); ++i)
   {
      delete m_rows(i);
   }
}

inline void    LinAlgFullCholesky::FullCholesky::FCMatrix::backSub(Vector<double> &b) const
{
   const int n = size();
   // Solve C*x = b, store the result in b
   for (int i = 1; i <= n; ++i)
   {
      const Row &row = *m_rows(i);
      for (int j = 1; j <= i - 1; ++j)
      {
         b(i) -= row(j) * b(j);
      }
      b(i) /= row(i); // Divide by diagonal element
   }
}

inline LinAlgFullCholesky::FullCholesky::FCTranspose::FCTranspose(int size)
: m_rows(size)
{
   const int n = size;
   for (int i = 1; i <= n; ++i)
   {
      m_rows(i) = new Row(n + 1 - i); // First row has length n, last row has length 1
   }
}

inline LinAlgFullCholesky::FullCholesky::FCTranspose::~FCTranspose()
{
   for (int i = 1; i <= m_rows.size(); ++i)
   {
      delete m_rows(i);
   }
}

inline void LinAlgFullCholesky::FullCholesky::FCTranspose::initiate(const FCMatrix &rhs)
{
   const int n = size();
   for (int i = 1; i <= n; ++i) // Loop over rows of 'rhs'
   {
      const Row &row = *rhs.m_rows(i);
      for (int j = 1; j <= i; ++j) // Loop over row 'i' in 'rhs'
      {
         // Found (i, j)-element in 'rhs' (row2). Add as (j, i)-element at us (row1)
         (*m_rows(j))(i - j + 1) = row(j);
      }
   }
}

inline void    LinAlgFullCholesky::FullCholesky::FCTranspose::backSub(Vector<double> &b) const
{
   const int n = size();
   // Solve C^T*x = b, store the result in b
   for (int i = n; i >= 1; --i)
   {
```

```
        Row &row = *m_rows(i);
        for (int j = n; j >= i + 1; --j)
        {
            b(i) -= row(j - i + 1) * b(j);
        }
        b(i) /= row(1); // Divide by diagonal element
    }
}

inline LinAlgFullCholesky::FullCholesky::FullCholesky(int size)
: m_matrix(size), m_transpose(size), m_dummy(size) {}

inline void    LinAlgFullCholesky::FullCholesky::initiate(const Matrix<double> &rhs)
{
    const int n = size();
    if ( (rhs.rows() != n) || (rhs.cols() != n) )
    {
        throw "Incompatible matrix dimensions in initiate"; // rhs must be n*n
    }
    for (int i = 1; i <= n; ++i)
    {
        Row &row = *m_matrix.m_rows(i);
        for (int j = 1; j <= i; ++j)
        {
            row(j) =  rhs(i, j);
        }
    }
}

// Performs overwriting Cholesky decompsition
inline void    LinAlgFullCholesky::FullCholesky::decompose()
{
    const int n = size(); // Matrix size
    for (int j = 1; j <= n; ++j)
    {
        Row &row1 = *m_matrix.m_rows(j); // Upper row
        // Compute c(j, j)
        double &cjj = row1(j); // To be computed
        for (int k = 1; k <= j - 1; ++k)
        {
            cjj -= row1(k) * row1(k);
        }
        if (cjj <= 0.0)
        {
            throw MatrixNotPositiveDefinite();
        }
        cjj = sqrt(cjj);
        // Compute c(i, j) for i > j
        for (int i = j + 1; i <= n; ++i)
        {
            Row &row2 = *m_matrix.m_rows(i); // Upper row
            double &cij = row2(j); // To be computed
            for (int k = 1; k <= j - 1; ++k)
            {
                cij -= row1(k) * row2(k);
            }
            cij /= cjj; // We know that cjj > 0 because of test above!
        }
    }
    m_transpose.initiate(m_matrix); // Initialize transpose of matrix A (used for backSub)
}

inline void    LinAlgFullCholesky::FullCholesky::backSub(Vector<double> &b) const
{
    const int n = size();
    if (b.size() != n)
    {
        throw "Incompatible matrix dimensions in backSub"; // b must be of order n
    }
    m_matrix.backSub(b);
```

```
    m_transpose.backSub(b);
}

inline void    LinAlgFullCholesky::FullCholesky::inverseDiag(Vector<double> &res) const
{
    const int n = size();
    if (res.size() != n)
    {
        throw "Incompatible matrix dimensions in inverseDiag"; // res must be of order n
    }
    for (int c = 1; c <= n; ++c) // Back substitute column c
    {
        // Initiate m_dummy to a zero vector, but with a one at position c
        m_dummy(c) = 1.0;
        for (int i = c + 1; i <= n; ++i)
        {
            m_dummy(i) = 0.0;
        }
        { // Solve A*x = m_dummy partly
            // First, solve C*y = m_dummy, store the result in m_dummy
            for (int i = c; i <= n; ++i) // Need not bother about i < c, because m_dummy is zero there!
            {
                const Row &row = *m_matrix.m_rows(i);
                for (int j = c; j <= i - 1; ++j)
                {
                    m_dummy(i) -= row(j) * m_dummy(j);
                }
                m_dummy(i) /= row(i); // Divide by diagonal element
            }
            // Next, solve C^T * x = y (= m_dummy now) partly
            for (int i = n; i >= c; --i)
            {
                Row &row = *m_transpose.m_rows(i);
                for (int j = n; j >= i + 1; --j)
                {
                    m_dummy(i) -= row(j - i + 1) * m_dummy(j);
                }
                m_dummy(i) /= row(1); // Divide by diagonal element
            }
        }
        res(c) = m_dummy(c); // The requested diagonal element
    }
}

inline void    LinAlgFullCholesky::FullCholesky::backSub_diag2lower(Matrix<double> &m) const
{
    const int n = size();
    if ( (m.rows() != n) || (m.cols() != n) )
    {
        throw "Incompatible matrix dimensions in backSub_diag2lower"; // m must be of order n x n
    }
    for (int c = 1; c <= n; ++c) // Back substitute column c
    {
        m_dummy(c) = m(c, c);
        for (int i = c + 1; i <= n; ++i)
        {
            m_dummy(i) = 0.0;
        }
        { // Solve A*x = m partly
            // First, solve C*y = m, store the result in m
            for (int i = c; i <= n; ++i) // Need not bother about rows < c, because m is zero there!
            {
                const Row &row = *m_matrix.m_rows(i);
                for (int j = c; j <= i - 1; ++j)
                {
                    m_dummy(i) -= row(j) * m_dummy(j);
                }
                m_dummy(i) /= row(i); // Divide by diagonal element
            }
            // Next, solve C^T * x = y (= m now) partly
```

```
            for (int i = n; i >= c; --i)
            {
                Row &row = *m_transpose.m_rows(i);
                for (int j = n; j >= i + 1; --j)
                {
                    m_dummy(i) -= row(j - i + 1) * m_dummy(j);
                }
                m_dummy(i) /= row(1); // Divide by diagonal element
            }
        }
        for (int i = c; i <= n; ++i)
        {
            m(i, c) = m_dummy(i);
        }
    }
}

// ------------------------------------------------------------------ SparseCholesky
inline LinAlgSparseCholesky::SparseCholesky::SCMatrix::SCMatrix(int size)
: m_rows(size)
{
    for (int i = 1; i <= size; ++i)
    {
        m_rows(i) = new Row(i);
    }
}

inline LinAlgSparseCholesky::SparseCholesky::SCMatrix::~SCMatrix()
{
    for (int i = 1; i <= m_rows.size(); ++i)
    {
        delete m_rows(i);
    }
}

inline void   LinAlgSparseCholesky::SparseCholesky::SCMatrix::backSub(Vector<double> &b) const
{
    const int n = size();

    // Solve C*x = b, store the result in b
    for (int i = 1; i <= n; ++i)
    {
        const Row &row = *m_rows(i);
        for (int p = 1; p < row.m_nItems; ++p)
        {
            b(i) -= row(p).m_val * b(row(p).m_j);
        }
        b(i) /= row(row.m_nItems).m_val; // Divide by diagonal element
    }
}

inline LinAlgSparseCholesky::SparseCholesky::SCTranspose::SCTranspose(int size)
: m_rows(size)
{
    const int n = size;
    for (int i = 1; i <= n; ++i)
    {
        m_rows(i) = new Row(n + 1 - i); // First row has length n, last row has length 1
    }
}

inline LinAlgSparseCholesky::SparseCholesky::SCTranspose::~SCTranspose()
{
    for (int i = 1; i <= m_rows.size(); ++i)
    {
        delete m_rows(i);
    }
}

inline void   LinAlgSparseCholesky::SparseCholesky::SCTranspose::backSub(Vector<double> &b) const
```

```
{
   const int n = size();

   // Solve C^T*x = b, store the result in b
   for (int i = n; i >= 1; --i)
   {
      Row &row = *m_rows(i);
      for (int p = row.m_nItems; p > 1; --p)
      {
         b(i) -= row(p).m_val * b(row(p).m_j);
      }
      b(i) /= row(1).m_val; // Divide by diagonal element
   }
}

inline void LinAlgSparseCholesky::SparseCholesky::SCTranspose::initiate(const SCMatrix &rhs)
{
   const int n = size();
   for (int i = 1; i <= n; ++i)
   {
      m_rows(i)->reset();
   }
   for (int i = 1; i <= n; ++i) // Loop over rows of 'rhs'
   {
      const Row &row2 = *rhs.m_rows(i);
      for (int p = 1; p <= row2.m_nItems; ++p) // Loop over row 'i' in 'rhs'
      {
         int j = row2(p).m_j;
         // Found (i, j)-element in 'rhs' (row2). Add as (j, i)-element at us (row1)
         Row &row1 = *m_rows(j);
         row1(++row1.m_nItems).set(i, row2(p).m_val);
      }
   }
}

inline LinAlgSparseCholesky::SparseCholesky::SparseCholesky(int size, double tol)
: m_matrix(size), m_transpose(size), m_newRow(size), m_tol(tol), m_dummy(size), m_firstIx(size) {}

inline void   LinAlgSparseCholesky::SparseCholesky::initiate(const Matrix<double> &rhs)
{
   const int n = size();
   if ( (rhs.rows() != n) || (rhs.cols() != n) )
   {
      throw "Incompatible matrix dimensions in initiate"; // rhs must be n*n
   }
   for (int i = 1; i <= n; ++i)
   {
      Row &row = *m_matrix.m_rows(i);
      for (int j = 1; j <= i; ++j)
      {
         row(j).set(j, rhs(i, j));
      }
      row.m_nItems = i;
   }
}

// Performs overwriting Cholesky decompsition of the sparse matrix A
inline void   LinAlgSparseCholesky::SparseCholesky::decompose()
{
   pack(); // Convert to sparse representation
   const int n = size(); // Matrix size
   for (int i = 1; i <= n; ++i)
   {
      Row &row = *m_matrix.m_rows(i);
      m_newRow.reset();
      for (int p = 1; p < row.m_nItems; ++p)
      {
         for (int j = row(p).m_j; j < row(p + 1).m_j; ++j)
         {
            decompose_computeCij(i, j, p);
```

```
            }
        }
        decompose_insertNewRow(i); // Important to insert new row before computing diagonal element!
        decompose_computeCii(i);
    }
    m_transpose.initiate(m_matrix); // Initialize transpose of matrix A (used for backSub)
}

// Converts full representation -> sparse representation
inline void LinAlgSparseCholesky::SparseCholesky::pack()
{
    const int n = size();
    for (int i = 1; i <= n; ++i)
    {
        Row &row = *m_matrix.m_rows(i);
        int p = 0; // Last filled p (= number of filled p's)
        for (int j = 1; j < i; ++j) // Loop over off-diagonal elements
        {
            if (fabs(row(j).m_val) > m_tol) // Non-zero element?
            {
                row(++p).set(j, row(j).m_val);
            }
        }
        row(++p).set(i, row(i).m_val); // Always include diagonal elements!
        row.m_nItems = p; // Number of non-zero elements in row
    }
}

// Computed the diagonal element (i, i) of the Cholesky decomposition C of the sparse matrix A: A = (
// Note that the new row has already been inserted, so we don't have to look for new elements.
inline void LinAlgSparseCholesky::SparseCholesky::decompose_computeCii(const int i)
{
    Row     &row    = *m_matrix.m_rows(i);   // Row i (old elements)
    int     p       = row.m_nItems;          // Points at last (the diagonal) entry of the row
    double  res     = row(p).m_val;   // The value to be computed. Start with the diagonal entry
    --p;
    while (p > 0)
    {
        res -= row(p).m_val * row(p).m_val;
        --p;
    }
    // Before drawing square root, check that res is positive
    if (res <= 0.0)
    {
        throw MatrixNotPositiveDefinite(); // Error! Matrix A was not positive definite
    }
    res = sqrt(res); // The final result
    row(row.m_nItems).m_val = res;
}

// Computes an off-diagonal element of the Cholesky decomposition C of the sparse matrix A: A = C * (
// (i, j) are the indeces of the element to be computed, j < i
// pParam is an index for row i. If the (i, j) entry exists in A, the pParam points at it. Otherwise,
//   pParam points at the first entry left of (i, j). Such an entry must exist, in other words it is
//   illegal (and unnecessary!) to call this method for entries left of the first non-zero entry in
inline void LinAlgSparseCholesky::SparseCholesky::decompose_computeCij(const int i, const int j, co
{
    Row     &row1       = *m_matrix.m_rows(j);   // Upper row (j)
    Row     &row2Old    = *m_matrix.m_rows(i);   // Lower row (i); entries that already existed in
    Row     &row2New    = m_newRow;              // Entries in lower row (j) that appear during the
    int     p1          = row1.m_nItems;         // Points at last (the diagonal) entry of row1
    double  cjj         = row1(p1).m_val;        // Upper diagonal value
    int     p2Old       = pParam;
    const bool  aijExists   = (row2Old(p2Old).m_j == j);   // Did the (i, j) entry already exist in A
    double  res         = 0.0;                   // The value to be computed
    if (aijExists)
    {
        res = row2Old(p2Old).m_val;
        --p2Old; // Start one step left of the entry if it existed in A
    }
```

```
int        p2New        = row2New.m_nItems;    // Last new element
bool    done        = false;
if (p2Old == 0)
{
   done = true; // No more elements to be processed
}
// Start computation. We know that p1>0, p2Old>0 and that "j1>j2" so that we start by decreasing p
// We do not know whether p2New>0.
while (!done)
{
   { // BLOCK 1, decrease p1
      int       j2;           // Largest j index in row 2
      bool    j2IsOld;    // Does j2 come from the old row?
      if ( (p2New > 0) && (row2New(p2New).m_j > row2Old(p2Old).m_j) ) // Compute j2 and j2IsOld
      {
         j2        = row2New(p2New).m_j;
         j2IsOld    = false;
      }
      else
      {
         j2        = row2Old(p2Old).m_j;
         j2IsOld    = true;
      }
      // Step p1 down to j2
      --p1; // We know that we must step at least one step!
      while ( (p1 > 0) && (row1(p1).m_j > j2) )
      {
         --p1;
      }
      // Now "p1 <= j2"
      if ( (p1 > 0) && (row1(p1).m_j == j2) ) // Match?
      {
         double val2 = j2IsOld ? row2Old(p2Old).m_val : row2New(p2New).m_val; // Value from row 2
         res -= row1(p1).m_val * val2; // Match found, modify res
         --p1; // So that "j1 < j2" and we can start decreasing the p2's
      }
      if (p1 == 0) // If we reached the end of row1, then quit
      {
         done = true;
      }
   } // End of BLOCK 1
   if (!done)
   { // BLOCK 2: decrease the p2's
      int j1 = row1(p1).m_j; // This is >0. Step the p2's down to j1
      while ( (p2Old > 0) && (row2Old(p2Old).m_j > j1) ) // Decrease p2Old
      {
         --p2Old;
      }
      while ( (p2New > 0) && (row2New(p2New).m_j > j1) )
      {
         --p2New;
      }
      // Now, both p2's are "<=j1". Any match?
      if ( (p2Old > 0) && (row2Old(p2Old).m_j == j1) ) // Old match?
      {
         res -= row1(p1).m_val * row2Old(p2Old).m_val;
         --p2Old;
      }
      else if ( (p2New > 0) && (row2New(p2New).m_j == j1) ) // New match?
      {
         res -= row1(p1).m_val * row2New(p2New).m_val;
         --p2New;
      }
      if (p2Old == 0) // If we reached the end of row2, then quit
      {
         done = true;
      }
   } // End of BLOCK 2
} // End of while (!done)
res /= cjj; // This is the computed value for cij
```

```
        if (aijExists) // If the entry already existed, just overwrite it
        {
            row2Old(pParam).m_val = res;
        }
        else if (fabs(res) > m_tol) // If the entry now appeared, create a new Item
        {
            Item &newItem = row2New(++row2New.m_nItems);
            newItem.set(j, res);
        }
    }

    // Decomposition of row i is done. Some new elements might have appeared, which have been stored in
    // This method inserts them into row i at the correct positions.
    // We insert from the right, so that no harmful overwriting takes place.
    inline void LinAlgSparseCholesky::SparseCholesky::decompose_insertNewRow(int i)
    {
        Row   &row1   = *m_matrix.m_rows(i);   // Old elements
        Row   &row2   = m_newRow;              // New elements
        int   p1      = row1.m_nItems;         // Points at last old element
        int p2      = row2.m_nItems;         // Points at last new element
        row1.m_nItems = p1 + p2;
        while (p2 > 0) // Keep inserting until there are no new elements left
        {
            if (row1(p1).m_j > row2(p2).m_j)
            {
                row1(p1 + p2) = row1(p1); // Copy data
                --p1;
            }
            else
            {
                row1(p1 + p2) = row2(p2); // Copy data
                --p2;
            }
        }
    }

    inline void LinAlgSparseCholesky::SparseCholesky::backSub(Vector<double> &b) const
    {
        const int n = size();
        if (b.size() != n)
        {
            throw "Incompatible matrix dimensions in backSub"; // b must be of order n
        }
        m_matrix.backSub(b);
        m_transpose.backSub(b);
    }

    inline void LinAlgSparseCholesky::SparseCholesky::inverseDiag(Vector<double> &res) const
    {
        const int n = size();
        if (res.size() != n)
        {
            throw "Incompatible matrix dimensions in inverseDiag"; // res must be of order n
        }

        for (int i = 1; i <= n; ++i)
        {
            m_firstIx(i) = 1; // The first item in row i that we need to bother about (>= c; the other item
        }
        for (int c = 1; c <= n; ++c) // Back substitute column c
        {
            // Initiate m_dummy to a zero vector, but with a one at position c
            m_dummy(c) = 1.0;
            for (int i = c + 1; i <= n; ++i)
            {
                m_dummy(i) = 0.0;
            }
            { // Solve A*x = m_dummy partly
                // First, solve C*y = m_dummy, store the result in m_dummy
                for (int i = c; i <= n; ++i) // Need not bother about i < c, because m_dummy is zero there!
```

```
            {
                const Row &row = *m_matrix.m_rows(i);
                for (int p = m_firstIx(i); p < row.m_nItems; ++p)
                {
                    m_dummy(i) -= row(p).m_val * m_dummy(row(p).m_j);
                }
                if (row(m_firstIx(i)).m_j == c)
                {
                    ++m_firstIx(i); // This entry will not be important at the next c value
                }
                m_dummy(i) /= row(row.m_nItems).m_val; // Divide by diagonal element
            }
            // Next, solve C^T * x = y (= m_dummy now) partly
            for (int i = n; i >= c; --i)
            {
                Row &row = *m_transpose.m_rows(i);
                for (int p = row.m_nItems; p > 1; --p)
                {
                    m_dummy(i) -= row(p).m_val * m_dummy(row(p).m_j);
                }
                m_dummy(i) /= row(1).m_val; // Divide by diagonal element
            }
        }
        res(c) = m_dummy(c); // The requested diagonal element
    }
}

inline void   LinAlgSparseCholesky::SparseCholesky::backSub_diag2lower(Matrix<double> &m) const
{
    const int n = size();
    if ( (m.rows() != n) || (m.cols() != n) )
    {
        throw "Incompatible matrix dimensions in backSub_diag2lower"; // m must be of order n x n
    }

    for (int i = 1; i <= n; ++i)
    {
        m_firstIx(i) = 1; // The first item in row i that we need to bother about (>= c; the other iten
    }
    for (int c = 1; c <= n; ++c) // Back substitute column c
    {
        m_dummy(c) = m(c, c);
        for (int i = c + 1; i <= n; ++i)
        {
            m_dummy(i) = 0.0;
        }
        { // Solve A*x = m partly
            // First, solve C*y = m, store the result in m
            for (int i = c; i <= n; ++i) // Need not bother about rows < c, because m is zero there!
            {
                const Row &row = *m_matrix.m_rows(i);
                for (int p = m_firstIx(i); p < row.m_nItems; ++p)
                {
                    m_dummy(i) -= row(p).m_val * m_dummy(row(p).m_j);
                }
                if (row(m_firstIx(i)).m_j == c)
                {
                    ++m_firstIx(i); // This entry will not be important at the next c value
                }
                m_dummy(i) /= row(row.m_nItems).m_val; // Divide by diagonal element
            }
            // Next, solve C^T * x = y (= m now) partly
            for (int i = n; i >= c; --i)
            {
                Row &row = *m_transpose.m_rows(i);
                for (int p = row.m_nItems; p > 1; --p)
                {
                    m_dummy(i) -= row(p).m_val * m_dummy(row(p).m_j);
                }
                m_dummy(i) /= row(1).m_val; // Divide by diagonal element
```

```
            }
        }
        for (int i = c; i <= n; ++i)
        {
            m(i, c) = m_dummy(i);
        }
    }
}
```