2005

Data Structures and Algorithms for Efficient Solution of Simultaneous Linear Equations from 3-D Ice Sheet Models

Rodney A. Jacobs

DATA STRUCTURES AND ALGORITHMS FOR EFFICIENT SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS FROM 3-D ICE SHEET MODELS

By

Rodney A. Jacobs

B.S. Massachusetts Institute of Technology, 1976

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Science)

The Graduate School

The University of Maine

December, 2005

Advisory Committee:

Dr. James L. Fastook, Professor of Computer Science, Advisor

Dr. Phillip M. Dickens, Assistant Professor of Computer Science

Dr. David Hiebeler, Assistant Professor of Mathematics

DATA STRUCTURES AND ALGORITHMS FOR EFFICIENT SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS FROM 3-D ICE SHEET MODELS

By Rodney A. Jacobs

Thesis Advisor: Dr. James L. Fastook

An Abstract of the Thesis Presented
In Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Science)
December, 2005

Two current software packages for solving large systems of sparse simultaneous linear equations are evaluated in terms of their applicability to solving systems of equations generated by the University of Maine Ice Sheet Model. SuperLU, the first package, has been developed by researchers at the University of California at Berkeley and the Lawrence Berkeley National Laboratory. UMFPACK, the second package, has been developed by T. A. Davis of the University of Florida who has ties with the U. C. Berkeley researchers as well as European researchers. Both packages are direct solvers that use LU factorization with forward and backward substitution.

The University of Maine Ice Sheet Model uses the finite element method to solve partial differential equations that describe ice thickness, velocity, and temperature throughout glaciers as functions of position and time. The finite element method generates systems of linear equations having tens of thousands of variables and one hundred or so non-zero coefficients per equation. Matrices representing these systems of equations may be strictly banded or banded with right and lower borders.

In order to efficiently interface the software packages with the ice sheet model, a modified compressed column data structure and supporting routines were designed and written. The data structure interfaces directly with both software packages and allows the ice sheet model to access matrix coefficients

by row and column number in roughly 100 nanoseconds while only storing non-zero entries of the matrix. No a priori knowledge of the matrix's sparsity pattern is required.

Both software packages were tested with matrices produced by the model and performance characteristics were measured and compared with banded Gaussian elimination. When combined with high performance basic linear algebra subprograms (BLAS), the packages are as much as 5 to 7 times faster than banded Gaussian elimination. The BLAS produced by K. Goto of the University of Texas was used. Memory usage by the packages varied from slightly more than banded Gaussian elimination with UMFPACK, to as much as a 40% savings with SuperLU. In addition, the packages provide componentwise backward error measures and estimates of the matrix's condition number. SuperLU is available for parallel computers as well as single processor computers. UMPACK is only for single processor computers. Both packages are also capable of efficiently solving the bordered matrix problem.

DEDICATION

To my wife Susie and daughter Rebecca.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. James Fastook, for his guidance and support. I would also like to thank Aitbala Sargent who integrated the routines developed in this work with The University of Maine Ice Sheet Model. This work was also made possible by J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li and all the developers of SuperLU; T. A. Davis, the developer of UMFPACK; and K. Goto, developer of the BLAS software.

My employer, N. H. Bragg & Sons, has provided the financial support and flexible working hours to make my graduate education at the University of Maine possible. They are a 150+ year-old company with the talent and vitality to constantly reinvent themselves and be a competitive, successful company. Thank you.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

1. Introduction 1

2. Mathematical Foundations for Solving Systems of Linear Equations 10
   2.1. Representing Systems of Linear Equations 10
   2.2. Solving Triangular Systems of Linear Equations 11
   2.3. Gaussian Elimination 13
   2.4. LU Factorization 15
   2.5. Cost of Solving Systems of Linear Equations 20
   2.6. Error In Computed Solutions 22
   2.7. Algorithm Stability 23
   2.8. Vector and Matrix Norms 26
   2.9. Checking the Stability of the Calculations 29
   2.10. Ill-Conditioned Problems 31
   2.11. Scaling 34
   2.12. Iterative Refinement 36
   2.13. Special Cases 37

3. Issues Regarding Sparse Systems of Linear Equations 40
   3.1. Data Storage Schemes 41
   3.2. Common Operations On Sparse Matrices and Vectors 50
      3.2.1. Addition of Sparse Vectors 50
      3.2.2. Inner Product of Sparse Vectors 55
   3.3. Conflicting Optimization Requirements for Data Structures 56
   3.4. Markowitz Cost: Row and Column Orderings for Optimized LU Factorization 57
   3.5. Minimum Degree Pivot Selection 61
   3.6. Banded Matrices 63
   3.7. Frontal Methods 67

4. BLAS: Basic Linear Algebra Subprograms 73

5. Software Packages for Solving Sparse Systems of Linear Equations 79
   5.1. SuperLU 85
   5.2. UMFPACK 95

6. Software Interface to the Ice Sheet Model 108
   6.1. Data Structures 108
   6.2. Implementation of Modified Compressed Column Data Structures 112
   6.3. Performance of Modified Compressed Column Routines 117
   6.4. Computing and Printing Error Measures 118
   6.5. Procedural Interface To SuperLU 119
   6.6. Procedural Interface to UMFPACK 122

7. Establishing a Basis For Performance Measures 124

8. Testing and Benchmarking SuperLU and UMFPACK 134
8.1. Verification Testing 134
8.2. Test Matrices 137
8.3. Initial Tests of BGAUSS 138
8.4. Initial Tests of BLU 139
8.5. Initial Tests of SuperLU 140
8.6. Initial Tests of UMFPACK 143
8.7. Detailed Test Results 145
8.7.1. Detailed Results: Systems without Pressure 146
8.7.2. Detailed Results: Systems with Pressure 151

9. Conclusions and Future Work 157

BIBLIOGRAPHY 160

APPENDICIES 161

Appendix 1. Modified Compressed Column Routines 162
A1.1. ccadd.f 162
A1.2. ccbase0.c 164
A1.3. ccbase1.c 164
A1.4. ccget.f 165
A1.5. ccinit.f 166
A1.6. ccparam.h 167
A1.7. ccput.f 168
A1.8. ccsqz.f 170
A1.9. cctest.f 171
A1.10. cczero.f 173
A1.11. matdump.c 174

Appendix 2. Error Measures Routine 176

Appendix 3. SuperLU Interface Routine and Demonstration Program 178
A3.1. demo1.f 178
A3.2. sluxsolve.c 180

Appendix 4. UMFPACK Interface Routine and Demonstration Program 186
A4.1. demo.f 186
A4.2. umfsolve.c 188

Appendix 5. Simple Banded Gaussian Elimination Program 192

Appendix 6. Banded Gaussian Elimination Routines 195
A6.1. bandparam.h 195
A6.2. bcopy.f 197
A6.3. berror.f 198
A6.4. bgauss.f 200
A6.5. bge.f 201
A6.6. binit.f 204
A6.7. bload.f 205
A6.8. bscale.f 207
A6.9. buser.f 208

Appendix 7. Banded LU Factorization Routines 213
A7.1. blu.f 213

A7.2. blufac.f 214
A7.3. brefine.f 216
A7.4. blusolve.f 218
A7.5. blustats.f 220

BIOGRAPHY OF THE AUTHOR 221

LIST OF TABLES

Table 6. 1. Calling parameters for `matdump`. 115
Table 6. 2. Performance of modified compressed column routines. 117

Table 7. 1. Banded matrix control array. 127
Table 7. 2. Banded matrix information array. 128
Table 7. 3. Banded Gaussian elimination routines. 128
Table 7. 4. Banded LU factorization routines. 131

Table 8. 1. Test results for solutions of Equation 8.3. 134
Table 8. 2. Test results for solution of Equation 8.1. 136
Table 8. 3. Test matrices from the ice sheet model. 137
Table 8. 4. Initial BGAUSS tests with m3d.20x20x5. 138
Table 8. 5. Initial BLU tests with m3d.20x20x5. 139
Table 8. 6. Initial SuperLU tests with m3d.20x20x5. 140
Table 8. 7. Initial SuperLU tests with m3dp.20x20x5. 142
Table 8. 8. Initial UMFPACK tests with m3d.20x20x5. 143
Table 8. 9. Initial UMFPACK tests with m3dp.15x15x5. 144

LIST OF FIGURES

Figure 1. 1. Components of the UMISM. 1
Figure 1. 2. Node numbering for a 2-D FEM rectangular grid. 3
Figure 1. 3. FEM grid overlaying satellite image of Antarctica. 4
Figure 1. 4. Scatter plot of non-zero entries in an ice sheet matrix. 6
Figure 1. 5. Scatter plot of non-zero entries in a matrix that includes pressures. 9

Figure 3. 1. Triplet storage using arrays. 41
Figure 3. 2. Structure for storing a triplet. 42
Figure 3. 3. Singly linked list of triplets. 43
Figure 3. 4. Doubly linked list of triplets. 44
Figure 3. 5. Compressed column format. 45
Figure 3. 6. Compressed row format. 47
Figure 3. 7. Linked row format. 47
Figure 3. 8. Linked row and column format. 48
Figure 3. 9. Linked row and column format with embedded row and column numbers. 49
Figure 3. 10. Addition of ordered sparse vectors with overwriting. 51
Figure 3. 11. Addition of ordered sparse vectors. 52
Figure 3. 12. Addition of unordered sparse vectors with overwriting. 53
Figure 3. 13. Addition of unordered sparse vectors. 54
Figure 3. 14. Inner product of ordered sparse vectors. 55
Figure 3. 15. Inner product of unordered sparse vectors. 56
Figure 3. 16. Sparsity pattern for matrix **A**. 58
Figure 3. 17. Swapping first and last rows of **A**. 58
Figure 3. 18. Swapping first and last columns of **A**. 59
Figure 3. 19. Swapping first and last rows and first and last columns of **A**. 59
Figure 3. 20. Non-optimal fill-in with minimum degree. 62
Figure 3. 21. A banded 9x9 matrix. 63
Figure 3. 22. Variable-band matrix. 66
Figure 3. 23. A triangulated FEM region. 67
Figure 3. 24. Assembly tree of frontal method. 71
Figure 3. 25. A rectangular FEM region. 72
Figure 3. 26. Assembly tree for multi-frontal method. 72

Figure 4. 1. BLAS Quick Reference Guide. 77

Figure 5. 1. Freely available software for linear algebra on the Web. 80
Figure 5. 2. Sample SuperLU output. 93
Figure 5. 3. Dense array for assembling a frontal matrix. 99
Figure 5. 4. Sample UMFPACK output. 104

Figure 6. 1. Modified compressed column format. 109
Figure 6. 2. Sample output of `matdump`. 116

Figure 7. 1. Output from `bg0.f`. 125
Figure 7. 2. Sample output of `bgauss.f`. 130
Figure 7. 3. Sample output of `blu.f`. 132

Figure 8. 1. Non-zeros in **L+U** in systems without pressure. 146
Figure 8. 2. Memory usage in systems without pressure. 147
Figure 8. 3. Floating point operations in systems without pressure. 148
Figure 8. 4. CPU time for all solvers in systems without pressure. 149
Figure 8. 5. CPU time for solvers with BLAS in systems without pressure. 150

Figure 8. 6. MegaFLOPs per second in systems without pressure. 151
Figure 8. 7. Non-zeros in **L+U** in system with pressure. 152
Figure 8. 8. Memory usage in systems with pressure. 153
Figure 8. 9. Floating point operations in systems with pressure. 154
Figure 8. 10. CPU time in systems with pressure. 155
Figure 8. 11. MegaFLOPS in systems with pressure. 156

1. Introduction

Dr. James Fastook, his colleagues, and his students have developed the University of Maine Ice Sheet Model (UMISM) over the past 15 years. The model predicts ice thickness, velocity, and temperature of glaciers as functions of position and time. Inputs to the model are climate conditions, giving temperatures and precipitation rates, and bed conditions, giving elevations and sliding characteristics. Figure 1.1 illustrates the components of the model and the interrelationships of the components in terms of their inputs and outputs.

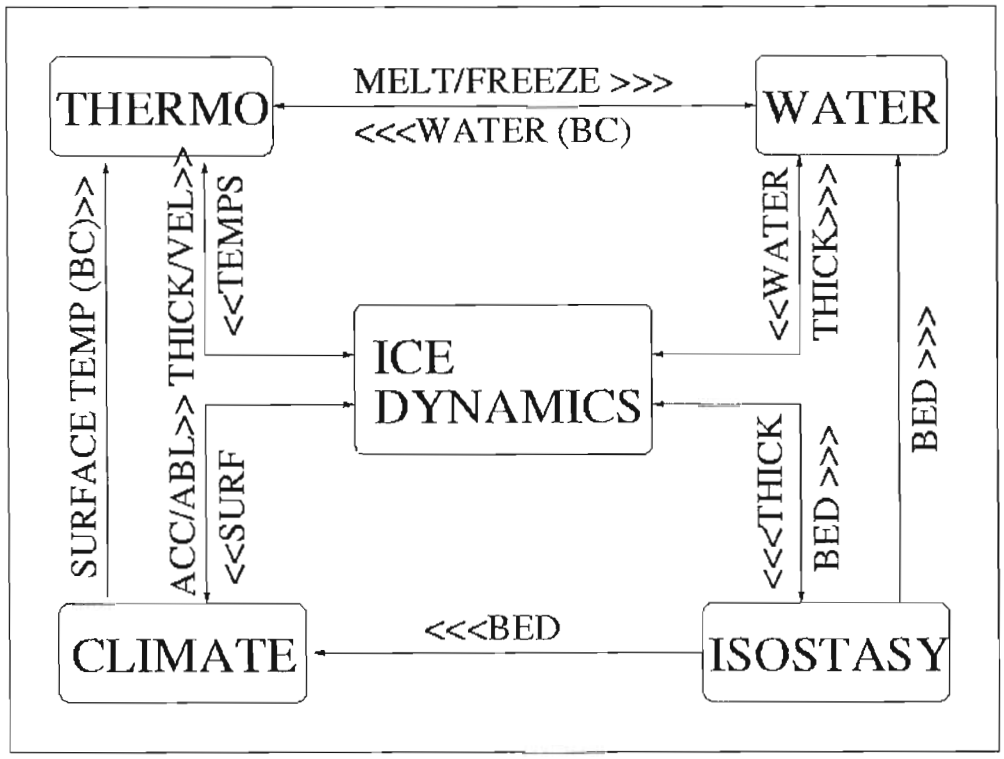

Figure 1. 1. Components of the UMISM.
Provided by James Fastook

The ice dynamics component is at the core of the model. It predicts ice thickness and ice velocity using

- Snowfall rates and melting rates generated by the climate module
- Ice temperatures generated by the thermodynamics module
- Presence or absence of water at the bed generated by the water module
- Bed elevation generated by the isostasy module.

The ice dynamics module uses ice temperature to determine how the ice reacts to the forces that stress it. Cold ice is harder than warm ice and is deformed at a slower rate. The ice dynamics module also uses the boundary condition characteristics between the ice and the earth. If the boundary condition is water, then the ice can slide without friction. At the opposite end of the spectrum, if the ice is frozen solid to the ground, then it does not slide. Finally, the weight of the ice depresses the ground, which lowers the surface elevation of the ice. The isostasy module computes the amount of bed depression.

Climate conditions at the surface of the ice depend upon surface elevation because temperature decreases with increasing altitude. The climate module uses the surface elevation generated by the ice dynamics and isostasy modules along with a climate model to predict surface temperatures, melting rates, and precipitation rates.

The thermodynamics module uses surface temperature as well as basal conditions and geothermal heating to compute temperature throughout the ice sheet. In addition, deformation of the ice due to movement also produces heat.

The water module uses bed characteristics from the isostasy module and basal temperatures to predict the presence of water.

The ice dynamics module uses partial differential equations (PDEs) derived from mass and momentum conservation principals as a basis for computing ice thickness and velocity. The thermodynamics module uses PDEs derived from energy conservation principals as a basis for computing ice temperatures. Combined with constitutive relationships that relate ice strain rates to temperature, and temperature to amount of heat, a complete system is formed for doing the fundamental calculations of ice thickness and velocity. The resulting PDEs are solved numerically using a mathematical technique known as the finite element method (FEM).

FEM computes the solution of the PDEs at discrete points in space and times. Figure 1.3 on the next page shows FEM spatial nodes overlaying a satellite image of Antarctica. This figure shows 4200 nodes separated by 70 kilometers over an area of 20 million square kilometers. This is a low resolution, 2-dimensional model. The colored contour lines are lines of constant elevation computed by the ice sheet model. The green square and magenta circle consist of nodes of a high resolution, embedded model that will be discussed shortly. While not fully shown in this figure, the region of calculation is chosen to include the entire ice sheet. Doing so ensures known boundary conditions: there is no ice at the boundary. The low resolution model typically uses 10 year time steps over a 100,000 year ice sheet cycle, giving a total of 10,000 time steps in a low resolution model run. The low resolution model is commonly run with 40 kilometer node spacing over Antarctica, giving a total of 16,000 nodes for the entire continent. With this node spacing, there are 16,000 ice thickness values and 32,000 velocity values for X and Y components of velocity. FEM does not require that nodes be arranged in rectangular grids, but this is the configuration commonly used in the ice sheet model.

FEM generates systems of simultaneous linear equations. The solutions of these equations give ice thickness and velocity. One equation is generated for each node and each degree of freedom. Ice thickness has one degree of freedom and 2-D velocities have two degrees of freedom. Non-zero contributions to an equation come from the node being considered and the nodes that are immediate neighbors of this node. Nodes are generally numbered sequentially starting along the coordinate axis with the least number of nodes. Figure 1.2 illustrates node numbering for a 3x4 grid.

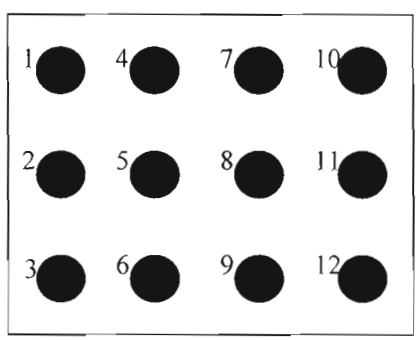

Figure 1. 2. Node numbering for a 2-D FEM rectangular grid.

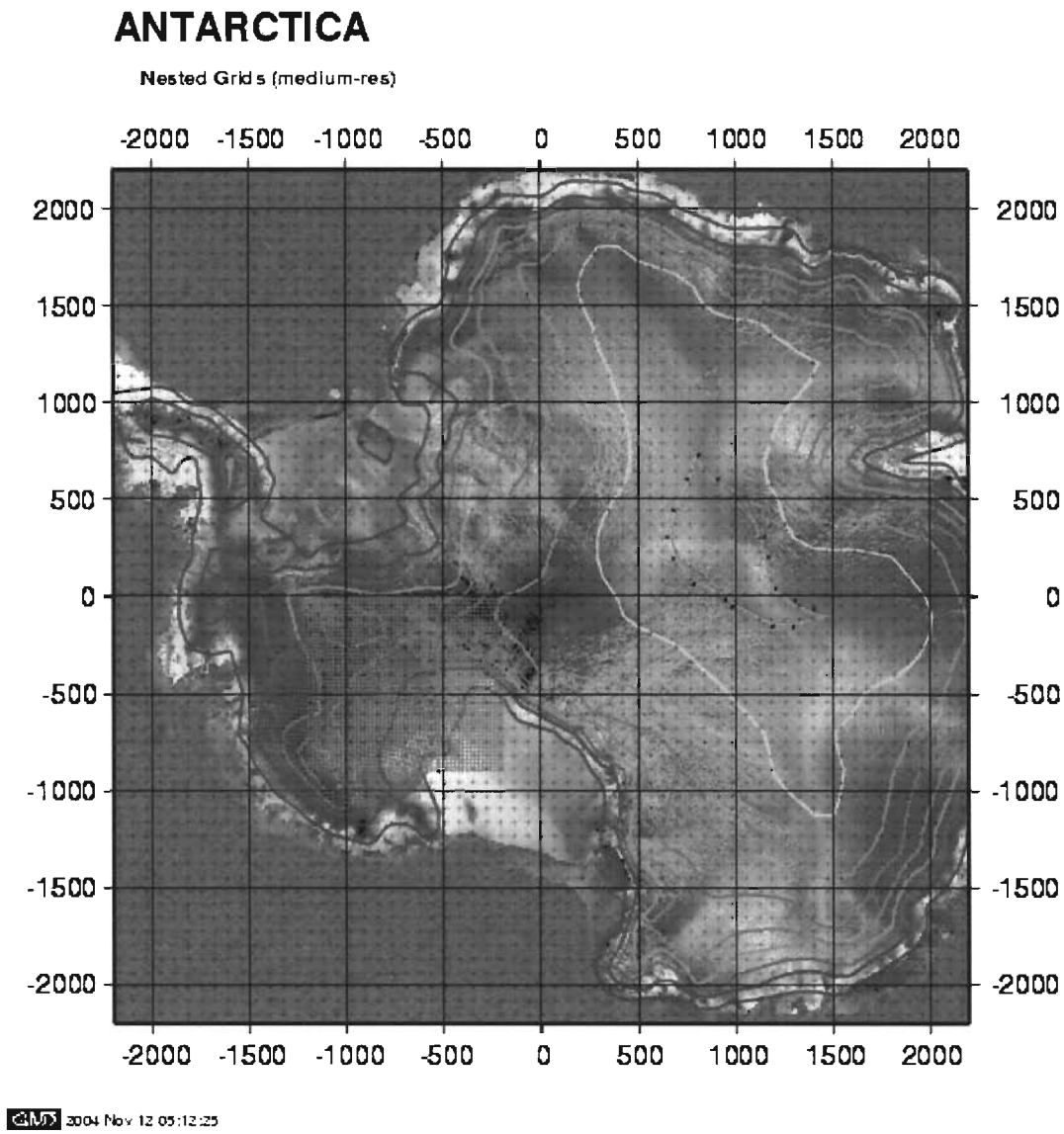

Figure 1. 3. FEM grid overlaying satellite image of Antarctica.
Provided by James Fastook.

The first equation for this grid, corresponding to node 1, will have non-zero contributions from nodes1, 2, 4, and 5. An equation for an interior node, such as node 5, will have 9 non-zero entries, 1 for itself and 8 from its immediate neighbors. For the 2-D model these linear equations have the mathematical property of being diagonally dominant. Diagonally dominant systems of linear equations can be easily solved using an iterative, numeric routine. The routine converges quickly for the ice sheet model. No working storage is required beyond storage for the non-zero coefficients and right hand side of the equations generated by FEM and storage for the solution vector. Chapter 2 contains additional information on diagonal dominance and iterative methods.

With a 40 kilometer grid spacing, the low resolution model does not provide as much resolution as desired in some areas where ice velocities can change rapidly with distance. One solution is to decrease the node spacing to get better resolution. However, the number of nodes is inversely proportional to the square of the node spacing. Reducing the node spacing by a factor of 4 increases the number of nodes 16 fold. Yet for the majority of the region, the higher resolution may not be necessary, resulting in much unnecessary computation. An alternative has been to embed a high resolution FEM grid within the low resolution grid over the area of interest. Results from the low resolution model provide boundary conditions that can be interpolated in space and time for the high resolution model. In practice, the low resolution model is run first and the results are saved. The model is then run for the high resolution grid with the saved low resolution results read and used as needed. The embedded model can use rectangular or curvilinear coordinates. These embedded models are illustrated in Figure 1.3.

Assumptions are made in the 2-D model that may be invalid in some regions. For this reason, the 3-D ice sheet model has been developed. The FEM method is still used to generate systems of simultaneous equations that are solved for ice thickness and velocity. However, the FEM grid is now three-dimensional and velocities have three degrees of freedom. In addition, the generated systems of equations are no longer diagonally dominant. For this reason it was decided that direct methods should be investigated for solving the equations generated by FEM. Direct methods use Gaussian elimination or similar tactics to solve

systems of simultaneous linear equations. The objective of this work is to investigate and implement direct methods for solving these systems of equations.

It is instructive to go into more detail regarding the 3-D model. First, consider the linear equations generated by FEM. The non-zero entries in an equation for a node are still generated by the node itself and its immediate neighbors. There are a total of 27 non-zero entries per equation when there is one degree of freedom. For 3-D velocities, there are 3 degrees of freedom for each node and 81 non-zero entries per equation. For a rectangular region that is 50x40x5 nodes, there are a total of 30,000 independent velocity variables and 30,000 equations, so the number of non-zero entries per equation is very small compared to the number of zero entries. Systems of equations that have many more zero entries than non-zero entries are called *sparse*, as opposed to *dense* systems that have many more non-zero entries than zero entries. If you look at the coefficients of the equations in the form of a square matrix, you find the diagonal elements of the matrix are non-zero. To the left and right of the diagonal are parallel bands of nonzero entries. Figure 1.4 is a scatter plot showing the non-zero entries from an actual matrix for ice velocities generated by the ice sheet model.

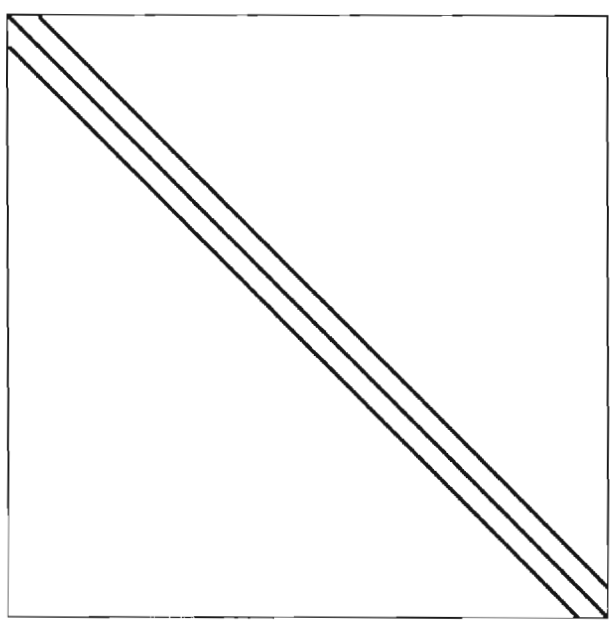

Figure 1. 4. Scatter plot of non-zero entries in an ice sheet matrix.

The number of entries between the outermost non-zero entries in a row of a banded matrix is called the bandwidth. The outermost non-zero entries are counted in the bandwidth. The number is important because the Gaussian elimination process generates non-zero entries within this band. The wider the bandwidth is, the more non-zero entries that are generated and the more arithmetic operations that must be performed. Minimizing bandwidth is a goal when performing Gaussian elimination on a banded matrix. The order in which nodes are numbered can have a dramatic affect on bandwidth. Suppose that the rectangular region is m by n by p nodes and that nodes are numbered along the m axis first, the n axis second, and the p axis third. For an interior node l with 1 degree of freedom, the bandwidth is the difference between the node numbers of the highest numbered node and lowest numbered node that are neighbors of l, plus 1. The lowest numbered neighbor of l is

$$l^- = l - mn - m - 1 \tag{1.1}$$

and the highest numbered neighbor of l is

$$l^+ = l + mn + m + 1 . \tag{1.2}$$

This gives the bandwidth with a single degree of freedom as

$$w_1 = l^+ - l^- + 1 = 2mn + 2m + 3 . \tag{1.3}$$

For three degrees of freedom the calculation is slightly different. Each node has three equations and three variables associated with it. Variables at a node are sequentially numbered first. Then the nodes along the m axis are numbered second, the nodes along the n axis third, and the nodes along the p axis fourth. There is one equation for each variable at each node. For each node and variable, non-zero contributions to an equation come from each variable of all the neighboring nodes as well as the variables of the node being considered. The lower bandwidth is the difference between the highest numbered variable of a node and the lowest numbered neighboring variable. For the highest numbered variable l of a node, the lower bandwidth is

$$w_{l3} = l - l_3^- = 3(mn + m + 1) + 2 . \tag{1.4}$$

The upper bandwidth is the difference between the lowest numbered variable of a node and the highest numbered neighboring variable. For the lowest numbered variable l of a node, the upper bandwidth is

$$w_{u3} = l_3^+ - l = 3(mn + m + 1) + 2 . \tag{1.5}$$

The total bandwidth with 3 degrees of freedom is then

$$w_3 = w_{l3} + w_{u3} + 1 = 6mn + 6m + 11 . \tag{1.6}$$

The bandwidths in equations 1.3 and 1.6 are minimized when m is chosen along the axis with the fewest nodes and n is chosen along the axis with the second fewest nodes.

For our 50x40x5 region with 3 degrees of freedom, the bandwidth is 1,241 and the number of equations is 30,000. With variables stored as 8-byte double precision real numbers and the equations solved using banded Gaussian elimination, 1,241*30,000*8 = 298 million bytes of storage are required to represent a matrix containing less than 2.43 million non-zero entries requiring 19.4 million bytes of storage. While banded Gaussian elimination is easy to implement, it is this disparity in storage sizes and the attendant number of arithmetic operations that prompts the question "Is there a better way to solve these systems of equations?"

There is an additional feature of the 3-D ice sheet model that also prompts the question for an alternative solution method. The version of the 3-D model that generates a banded system of equations eliminates internal pressures within the ice sheet. When these pressures are explicitly included in the model, the system of equations is no longer banded. Figure 1.5 shows a scatter plot of the non-zero entries in a matrix with explicitly specified pressures. Without the banded matrix structure, storing the matrix as a two dimensional array and straightforwardly applying Gaussian elimination is impractical.

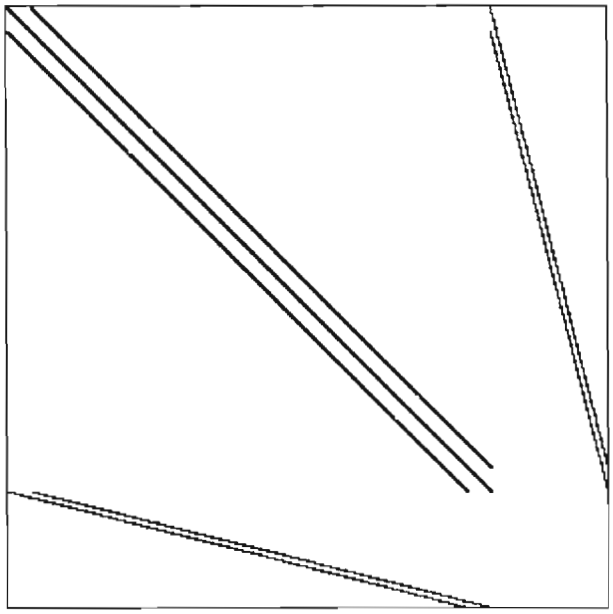

Figure 1. 5. Scatter plot of non-zero entries in a matrix that includes pressures.

Solving large systems of sparse simultaneous linear equations is a common need in science and engineering problems. Over the years a great deal of work has been done in this area by researchers [10][11]. Software to solve these problems has been developed and is readily available. In this work two current software packages were chosen and evaluated with respect to the ice sheet problem. One is SuperLU [8]. The other is UMFPACK [6]. In order to use these packages effectively the mathematics of solving these problems must be understood. In addition, efficient procedures and data structures are needed to interface the ice sheet model with the packages. Finally, the packages themselves must be understood to a level that enables us to use them properly. Each of these areas is addressed by this work. In addition, performance testing is done with banded Gaussian elimination serving as a benchmark.

2. Mathematical Foundations for Solving Systems of Linear Equations

A rudimentary understanding of the mathematical issues regarding systems of linear equations was essential to this work. Although some concepts of linear algebra were familiar to me from various science and engineering courses, I lack formal training in linear algebra and numerical analysis. In hindsight, having formal training in these areas would have made this work easier.

This chapter presents fundamental mathematical concepts required for understanding the solution process. Most of this information comes from Duff, et. al. [10], Golub and Van Loan [11], and Lay [12]. Most of the major concepts are motivated through discussion, but careful proofs are not generally given. The cited references provide further details.

2.1. Representing Systems of Linear Equations

A system of linear equations can be specified explicitly.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{12}x_3 = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \end{cases} \tag{2.1}$$

The a_{ij} and b_i terms are numeric constants and the x_i terms are variables. Solving the set of equations means finding a set of values for x_i such that all the equations are satisfied. The number of equations and the number of variables can be any whole number greater than zero. These numbers need not be equal as in this example. However, for ice sheet modeling, they are equal, and we will assume them to be equal throughout this work. The ice sheet model generates systems with thousands or tens of thousands of equations and variables.

Alternatively, equations (2.1) may be represented as a matrix equation.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{2.2}$$

The a_{ij} terms may be represented as a matrix **A**, and the x_i and b_i terms may be represented as vectors **x** and **b** respectively, so equations (2.1) may be represented even more compactly.

$$\mathbf{Ax} = \mathbf{b} \tag{2.3}$$

The a_{ii} entries of **A** are diagonal entries. In a square matrix, the diagonal entries lie along a diagonal line from the upper left corner of the matrix to the lower right corner. The set of diagonal entries is called the diagonal of the matrix.

A fourth representation of a system of linear equations is the summation notation.

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \qquad 1 \le i \le m \tag{2.4}$$

In this notation m is the number of equations and n is the number of variables.

2.2. Solving Triangular Systems of Linear Equations

A system of linear equations **Lx** = **b** is lower triangular when all entries of **L** above the diagonal have the value zero. Specifically, $a_{ij} = 0$ for $i < j$.

$$\begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{2.5}$$

This system is easily solved if the diagonal entries of **L** are non-zero.

$$\begin{cases} x_1 = b_1 / l_{11} \\ x_2 = (b_2 - l_{21} x_1) / l_{22} \\ \text{and in general for } i > 1: \\ x_i = (b_i - \sum_{j=1}^{i-1} l_{ij} x_j) / l_{ii} \end{cases} \tag{2.6}$$

This method is called *forward substitution*. The value of x_i is calculated using the previously calculated values of x_j for $1 \le j \le i\text{-}1$.

If

$$l_{ii} = 0 \tag{2.7}$$

and

$$b_i - \sum_{j=1}^{i-1} l_{ij} x_j \neq 0 \tag{2.8}$$

then x_i is undefined and the system has no solution. If

$$l_{ii} = 0 \tag{2.9}$$

and

$$b_i - \sum_{j=1}^{i-1} l_{ij} x_j = 0 \tag{2.10}$$

then x_i may have any value.

A system of linear equations **Ux** = **b** is upper triangular when all entries of **U** below the diagonal have the value zero. Specifically, $a_{ij} = 0$ for $i > j$.

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \tag{2.11}$$

This system is also easily solved if the diagonal entries of **U** are non-zero. Assume the system has n equations.

$$\begin{cases} x_n = b_n / u_{nn} \\ x_{n-1} = (b_{n-1} - u_{n-1,n} x_n) / u_{n-1,n-1} \\ \text{and in general for } i < n: \\ x_i = (b_i - \sum_{j=i+1}^{n} u_{ij} x_j) / u_{ii} \end{cases} \tag{2.12}$$

This method is call *backward substitution*. The value of x_i is calculated using the previously calculated values of x_j for $i+1 \leq j \leq n$. Like lower triangular systems, similar arguments can be made regarding the solution of the system when $u_{ii} = 0$.

2.3. Gaussian Elimination

Gaussian elimination is a method for transforming a system of linear equations to upper triangular form while preserving the value of the solution vector. Once the equations have been transformed to upper triangular from, backward substitution can be used to solve them. Gaussian elimination repeatedly applies three elementary transformations to the system that preserve the solution. These elementary transformations are:

1. Multiplying both sides of an equation by a constant.
2. Replacing an equation by the sum of itself and a multiple of another equation.
3. Interchanging the positions of two equations.

Consider this system of equations.

$$\left(\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right) \tag{2.13}$$

For simplicity, the variables x_i have been removed as have the arithmetic operators and relations. The right hand sides of the equations have been combined with the coefficients to produce a single augmented matrix. The first step of the elimination process uses the first equation to eliminate the coefficients in the first column below the diagonal. If $a_{11} \neq 0$, then multiplying the first equation by $-a_{21} / a_{11}$ and adding it to the second equation produces

$$\left(\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & b_2^{(2)} \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right) \tag{2.14}$$

where

$$a_{22}^{(2)} = a_{22} - (a_{21} / a_{11}) a_{12}\,, \tag{2.15}$$

$$a_{23}^{(2)} = a_{23} - (a_{21} / a_{11}) a_{13}\,, \tag{2.16}$$

$$b_2^{(2)} = b_2 - (a_{21} / a_{11}) b_1\,. \tag{2.17}$$

Similarly, multiplying the first equation by $-a_{31} / a_{11}$ and adding it to the third equation produces

$$\left(\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & b_2^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} & b_3^{(2)} \end{array}\right) \tag{2.18}$$

where

$$a_{32}^{(2)} = a_{32} - (a_{31} / a_{11}) a_{12}, \tag{2.19}$$

$$a_{33}^{(2)} = a_{33} - (a_{31} / a_{11}) a_{13}, \tag{2.20}$$

$$b_3^{(2)} = b_3 - (a_{31} / a_{11}) b_1. \tag{2.21}$$

If $a_{11} = 0$, then the first equation is interchanged with an equation that has a non-zero coefficient in the first column before the elimination calculations are performed. Interchanging equations is called *pivoting*. If all the coefficients in the first column are zero, then there is nothing to do in the first elimination step.

The second step of the elimination process uses the second equation to eliminate the coefficients below the diagonal in the second column. The second row is interchanged with the row below it if $a_{22}^{(2)} = 0$ and $a_{32}^{(2)} \neq 0$. Otherwise, the second row is multiplied by $-a_{32}^{(2)} / a_{22}^{(2)}$ and added to the third row to produce

$$\left(\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & b_2^{(2)} \\ 0 & 0 & a_{33}^{(3)} & b_3^{(3)} \end{array}\right) \tag{2.22}$$

where

$$a_{33}^{(3)} = a_{33}^{(2)} - \left(a_{32}^{(2)} / a_{22}^{(2)}\right) a_{23}^{(2)}, \tag{2.23}$$

$$b_3^{(3)} = b_3^{(2)} - \left(a_{32}^{(2)} / a_{22}^{(2)}\right) b_2^{(2)}. \tag{2.24}$$

The transformed set of equations now has the form

$$\mathbf{Ux} = \mathbf{c} \tag{2.25}$$

where

$$\mathbf{U} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(3)} \end{pmatrix} \tag{2.26}$$

and

$$\mathbf{c} = \begin{pmatrix} b_1 \\ b_2^{(2)} \\ b_3^{(3)} \end{pmatrix} \tag{2.27}$$

When there are more than three equations, the process is continued using the ith equation to eliminate coefficients below the diagonal in the ith column and using row interchanges to ensure $a_{ii}^{(i)} \neq 0$. The entries $a_{ii}^{(i)}$ are called *pivots*.

2.4. LU Factorization

Suppose $\mathbf{A} = \mathbf{LU}$ where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix. Then $\mathbf{Ax} = \mathbf{b}$ is equivalent to

$$\mathbf{LUx} = \mathbf{b}. \tag{2.28}$$

We can easily solve equation (2.28) using forward and backward substitution. Let

$$\mathbf{Ux} = \mathbf{c}. \tag{2.29}$$

Then

$$\mathbf{Lc} = \mathbf{b}. \tag{2.30}$$

We can solve $\mathbf{Lc} = \mathbf{b}$ for $\mathbf{c}$ using forward substitution. Given $\mathbf{c}$, we can solve $\mathbf{Ux} = \mathbf{c}$ for $\mathbf{x}$ using backward substitution. Furthermore, $\mathbf{Ax} = \mathbf{b}$ can be solved for multiple values of $\mathbf{b}$ by using the $\mathbf{L}$ and $\mathbf{U}$ factors of $\mathbf{A}$. There is no need to perform Gaussian elimination for each $\mathbf{b}$.

To compute the **LU** factorization, we can apply elementary transformations to **A** that are similar to the elementary Gaussian elimination transformations, thus transforming **A** into an upper triangular matrix that we can identify with **U**. Each elementary transformation consists of left multiplying A by an elementary transformation matrix. The end result is the matrix equation

$$\mathbf{E}_p\mathbf{E}_{p-1}\ldots\mathbf{E}_2\mathbf{E}_1\mathbf{A} = \mathbf{U} \tag{2.31}$$

The elementary transformation matrices are easily inverted, allowing us to move them from the left side of the equation to the right side. The inverted matrices are lower triangular matrices, and the product of lower triangular matrices is a lower triangular matrix. Thus the product of the inverted transformation matrices is **L**. If row interchanges are required to ensure that pivots are not zero, the resulting **LU** factorization is for **A** with the same row interchanges. Let's take a look at each of these statements in greater detail to understand how the process works. We'll begin by assuming all pivots are non-zero, so row interchanges are not required.

The first elementary transformation is adding a multiple of one row to a lower row in the matrix. This is similar to the second elementary transformation in Gaussian elimination. Like Gaussian elimination, this transformation does most of the work. For example, the elementary matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ k & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

adds k times the first row of **A** to the second row of **A**.

$$\begin{pmatrix} 1 & 0 & 0 \\ k & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ (a_{21}+ka_{11}) & (a_{22}+ka_{12}) & (a_{23}+ka_{13}) \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \tag{2.32}$$

The following matrix multiplications eliminate a_{21} and a_{31} from **A**

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -l_{31} & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ -l_{21} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} \end{pmatrix} \tag{2.33}$$

where $l_{21} = a_{21} / a_{11}$ and $l_{31} = a_{31} / a_{11}$.

Finally, we can eliminate $a_{32}^{(2)}$, obtaining **U**.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -l_{32} & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -l_{31} & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ -l_{21} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} \\ 0 & 0 & a_{33}^{(3)} \end{pmatrix} = \mathbf{U} \tag{2.34}$$

In general,

$$l_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}. \tag{2.35}$$

The inverses of the transformation matrices are

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -l_{32} & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & l_{32} & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -l_{31} & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ l_{31} & 0 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 \\ -l_{21} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.36}$$

The product of the inverted transformation matrices in reverse order gives

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix}. \tag{2.37}$$

The second elementary transformation is interchanging rows to avoid zero pivots. A matrix that interchanges rows or columns of another matrix when the matrices are multiplied is called a *permutation matrix*. Left multiplying a matrix by a permutation matrix interchanges rows. Right multiplying a matrix by a permutation matrix interchanges columns. For example, left multiplying **A** by the following permutation matrix swaps the second and third rows.

$$\mathbf{PA} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{31} & a_{32} & a_{33} \\ a_{21} & a_{22} & a_{22} \end{pmatrix} \tag{2.38}$$

When implemented on a computer with **A** represented as a two dimensional array of entries, the **LU** factorization can be done in place with the computed **L** and **U** factors replacing **A**. When this is done, the diagonal of **L**, which is 1's, is not explicitly stored; only the diagonal of **U** is stored. When pivoting is required, all **L** and **U** entries in the pivot row are interchanged with entries from a row lower in the matrix. In the general case, this algorithm produces an upper triangular matrix that looks like

$$\bar{\mathbf{L}}^{n-1}\mathbf{P}^{n-1}\bar{\mathbf{L}}^{n-2}\mathbf{P}^{n-2}\ldots\bar{\mathbf{L}}^{2}\mathbf{P}^{2}\bar{\mathbf{L}}^{1}\mathbf{P}^{1}\mathbf{A} = \mathbf{U}\,. \tag{2.39}$$

where $\mathbf{P}^k$ are elementary permutation matrices and $\bar{\mathbf{L}}^k$ are the product of elementary matrices for eliminating entries a_{ik} for $i > k$. Using the transformation

$$\bar{\mathbf{L}}^k = \mathbf{P}^{k+1}\mathbf{P}^{k+2}\ldots\mathbf{P}^{n-1}\bar{\bar{\mathbf{L}}}^k\mathbf{P}^{n-1}\mathbf{P}^{n-2}\ldots\mathbf{P}^{k+1} \qquad 1 \le k < n-1 \tag{2.40}$$

gives

$$\bar{\bar{\mathbf{L}}}^{n-1}\bar{\bar{\mathbf{L}}}^{n-2}\ldots\bar{\bar{\mathbf{L}}}^{2}\bar{\bar{\mathbf{L}}}^{1}\mathbf{P}^{n-1}\mathbf{P}^{n-2}\ldots\mathbf{P}^{2}\mathbf{P}^{1}\mathbf{A} = \mathbf{U}\,. \tag{2.41}$$

To verify this result, substitute for $\bar{\mathbf{L}}^1$ in equation (40) first. The elementary permutation matrices have the property $\mathbf{P}^i\mathbf{P}^i = \mathbf{I}$ where **I** is the identity matrix. This simply states that interchanging the same two rows twice produces the original matrix. Use this fact to simplify the result. Substitute for $\bar{\mathbf{L}}^2$ and simplify. Continue this process through $\bar{\mathbf{L}}^{n-1}$. Also observe that

$$\bar{\bar{\mathbf{L}}}^k = \mathbf{P}^{n-1}\mathbf{P}^{n-2}\ldots\mathbf{P}^{k+2}\mathbf{P}^{k+1}\bar{\mathbf{L}}^k\mathbf{P}^{k+1}\mathbf{P}^{k+2}\ldots\mathbf{P}^{n-2}\mathbf{P}^{n-2}\,. \tag{2.42}$$

The net result of these permutations on $\bar{\mathbf{L}}^k$ is to make $\bar{\bar{\mathbf{L}}}^k$ a lower unit triangular matrix with zero entries preserved and the same entries in column k as $\bar{\mathbf{L}}^k$ but with the entries reordered by the row interchanges.

The product of the elementary permutation matrices and **A** can be rewritten as the product of a combined permutation matrix and **A**.

$$\mathbf{P}^{n-1}\mathbf{P}^{n-2}\ldots\mathbf{P}^{2}\mathbf{P}^{1}\mathbf{A} = \mathbf{PA} \tag{2.43}$$

Thus, equation (2.42) can be rewritten as

$$\mathbf{PA} = \mathbf{LU}\,. \tag{2.44}$$

In practice, **P** is recorded as the factorization is computed and rows are interchanged. When solving the system of equations $\mathbf{Ax} = \mathbf{b}$ with **LU** factorization, we actually solve

$$\mathbf{LUx} = \mathbf{Pb}\,. \tag{2.45}$$

The row interchanges of **b** are consistent with row interchanges that would take place in Gaussian elimination.

As will be seen later, it is common to permute the columns of **A** when performing **LU** factorization for sparse matrices. When this is done, we have

$$\mathbf{PAQ} = \mathbf{LU} \tag{2.46}$$

where **Q** is a column permutation matrix. In general, all permutation matrices have the property $\mathbf{QQ}^{\mathrm{T}} = \mathbf{I}$ where $\mathbf{Q}^{\mathrm{T}}$ is the transpose of **Q**. Therefore, the original system of equations can be written as

$$\mathbf{PAQQ}^{\mathrm{T}}\mathbf{x} = \mathbf{Pb} \tag{2.47}$$

or

$$\mathbf{LUQ}^{\mathrm{T}}\mathbf{x} = \mathbf{Pb}\,. \tag{2.48}$$

We can solve this equation as follows. Setting $\mathbf{UQ}^{\mathrm{T}}\mathbf{x} = \mathbf{c}$, we can solve $\mathbf{Lc} = \mathbf{Pb}$ for **c** using forward substitution. Setting $\mathbf{Q}^{\mathrm{T}}\mathbf{x} = \mathbf{w}$ and knowing **c**, we can solve $\mathbf{Uw} = \mathbf{c}$ for **w** using backward substitution. Finally, we can compute $\mathbf{x} = \mathbf{Qw}$.

2.5. Cost of Solving Systems of Linear Equations

The cost of solving a system of linear equations can be measured as the number of arithmetic operations required to solve them. With Gaussian elimination we must count the number of arithmetic operations to transform the system of equations to an upper triangular system and the number of operations to find the solution using backward substitution. The number of operations to transform the system to an upper triangular system has two components: operations on the left hand sides of the equations and operations on the right hand sides. Looking at the left hand side first, consider the number of operations required to use row k of $\mathbf{A}^{(k)}$ to eliminate a_{ik} of row i. First, the ratio $l_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}$ must be formed. Then $a_{ij}^{(k+1)} = a_{ij}^{(k)} - a_{kj}^{(k)} l_{ik}$ must be formed for $k+1 \le j \le n$, giving a total of $1 + 2(n-k)$ operations. This set of operations must be performed $n-k$ times, giving a total of $(n-k) + 2(n-k)^2$ operations for $\mathbf{A}^{(k)}$. These operations must be performed for each $\mathbf{A}^{(k)}$ where $1 \le k \le n$, giving

$$\alpha_l = \sum_{k=1}^{n} (n-k) + 2(n-k)^2 \tag{2.49}$$

as the total number of operations to the left hand sides of the equations. Recognizing

$$\sum_{k=1}^{n} k = \frac{1}{2}n^2 + \frac{1}{2}n \tag{2.50}$$

and

$$\sum_{k=1}^{n} k^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n \tag{2.51}$$

gives

$$\alpha_l = \frac{2}{3}n^3 - \frac{1}{2}n^2 - \frac{1}{6}n\,. \tag{2.52}$$

On the right hand side, the number of operations is

$$\alpha_r = \sum_{k=1}^{n} 2(n-k) \tag{2.53}$$

or

$$\alpha_r = n^2 - n\,. \tag{2.54}$$

The number of operations to perform backward substitution is given by

$$\alpha_b = \sum_{k=1}^{n} (1 + 2(n-k)) \tag{2.55}$$

or

$$\alpha_b = n^2\,. \tag{2.56}$$

The total number operations to perform Gaussian elimination is the sum of these three components, giving

$$\alpha_g = \frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n \tag{2.57}$$

or

$$\alpha_g = O(n^3)\,. \tag{2.58}$$

For **LU** factorization, α_l operations are required to do the factorization. Performing forward substitution with a unit lower triangular system takes α_r operations, and performing backward substitution requires α_b operations. So the total number of operations is the same as Gaussian elimination.

These cost calculations are based on dense matrices. When dealing with sparse matrices, our goal is to achieve better performance.

2.6. Error In Computed Solutions

In addition to computing a solution to a system of equations, we must also evaluate the accuracy of the computed solution. Real numbers on a computer are generally represented in single precision or double precision floating point format. Single precision numbers have about 6 decimal digits of precision, while double precision numbers have about 16 decimal digits. These formats are unable to represent real numbers exactly. As computations are performed, we must concern ourselves with round off error and the evolving accuracy. As an example, computer addition is not associative. On a 3-digit computer, (100. + (.4 + (.4 + .4))) = 101., but (((100. + .4) + .4) + .4) = 100. From the previous section, solving a system of 10^4 equations will involve on the order of 10^{12} arithmetic operations. In addition to round off error, there is likely to be uncertainty in the values of **A** and **b** that must also be taken into account.

In theory, we should be able to put error bounds on the computations by following the sequence of operations performed by the algorithm used to solve the system of equations. In practice this approach tends to grossly overstate the errors that are actually observed because a portion of the round off error is reduced due to cancellation. Instead, the standard practice is to answer the two following questions [10].

1. Is the computed solution $\bar{\mathbf{x}}$ the exact solution of a nearby problem?
2. If small changes are made to the given problem, are changes to the exact solution also small?

A problem $\bar{\mathbf{A}}\mathbf{x} = \bar{\mathbf{b}}$ is considered nearby $\mathbf{A}\mathbf{x} = \mathbf{b}$ when small perturbations to **A** and **b** produce $\bar{\mathbf{A}}$ and $\bar{\mathbf{b}}$. When the first question is answered yes, the computational error has been kept under control. An algorithm that satisfies this property is called *stable*. When the algorithm is stable, it is as though we made small perturbations to the problem and solved the perturbed system exactly.

If the answer to the second question is yes, then the problem is called *well-conditioned*. If the problem is well-conditioned and the algorithm is stable, then our calculated solution is a good estimate of the exact solution. If the answer to the second question is no, then the problem is called *ill-conditioned*. If a problem

is ill-conditioned, then our solution is likely to have a large error even if the algorithm used to compute it is stable. The condition of a problem is a property of the problem.

The following sections will illustrate these concepts and present the mathematics for dealing with them in a concrete way.

2.7. Algorithm Stability

The following example from Duff, et.al. [10] demonstrates that Gaussian elimination and **LU** factorization can be unstable. Consider the following system of two equations on a computer that maintains 3 decimal digits of precision.

$$\begin{pmatrix} 0.001 & 2.42 \\ 1.00 & 1.58 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5.20 \\ 4.57 \end{pmatrix} \tag{2.59}$$

$l_{21} = 1000$, giving $a_{22}^{(2)} = 1.58 - 1000 * 2.42 = -2420$ and $b_2^{(2)} = 4.57 - 1000 * 5.20 = -5200$. Thus, the system transformed to an upper triangular system of equations is

$$\begin{pmatrix} 0.001 & 2.42 \\ 0 & -2420 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5.20 \\ -5200 \end{pmatrix}. \tag{2.60}$$

Using backward substitution to solve for **x**, we obtain

$$\bar{\mathbf{x}} = \begin{pmatrix} 0.00 \\ 2.15 \end{pmatrix} \tag{2.61}$$

while the correct solution to 3 decimal places is

$$\mathbf{x} \cong \begin{pmatrix} 1.18 \\ 2.15 \end{pmatrix}. \tag{2.62}$$

While x_2 has been accurately computed, x_1 has not. The same problem is manifested in **LU** factorization.

$$\begin{pmatrix} 0.001 & 2.42 \\ 1.00 & 1.58 \end{pmatrix} \cong \begin{pmatrix} 1 & \\ 1000 & 1 \end{pmatrix}\begin{pmatrix} 0.001 & 2.42 \\ & -2400 \end{pmatrix} \tag{2.63}$$

Computing $\mathbf{H} = \overline{\mathbf{L}}\overline{\mathbf{U}} - \mathbf{A}$ where $\mathbf{H}$ is the perturbation in $\mathbf{A}$ due to calculation error gives

$$\mathbf{H} = \begin{pmatrix} 0.00 & 0.00 \\ 0.00 & -1.58 \end{pmatrix}. \tag{2.64}$$

$h_{22} = -a_{22}$, so the perturbation due to calculation error is not small.

Another way to demonstrate the stability of these algorithms is to compute $\mathbf{r} = \mathbf{b} - \mathbf{A}\bar{\mathbf{x}}$. $\mathbf{r}$ is called the *residual*. From the calculations above,

$$\mathbf{r} = \begin{pmatrix} -.003 \\ 1.17 \end{pmatrix}. \tag{2.65}$$

r_2 is not small compared to $\bar{x}_2$, so again the algorithms appear unstable.

In each algorithm, the value of $l_{21} = 1000$ caused a_{22} to get lost in the growth of $a_{22}^{(2)}$. Duff, et. al. report that work done by Wilkinson and extended by Reid give the inequality

$$\left| h_{ij} \right| \le 5.01\varepsilon \; n \max_k \left| a_{ij}^{(k)} \right| \tag{2.66}$$

where ε is the relative precision of the computer (0.0005 for our hypothetical 3-digit computer) and n is the number of equations. This states that if the growth in $a_{ij}^{(k)}$ is kept small, the perturbation to $\mathbf{A}$ will be small. This suggests that if we swap the order of the equations in our sample system, l_{21} will be small, and the resulting accuracy of our solution should be good. The upper triangular form of the swapped equations is

$$\begin{pmatrix} 1.00 & 1.58 \\ 0.00 & 2.42 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4.57 \\ 5.20 \end{pmatrix} \tag{2.67}$$

Using backward substitution, we obtain the solution

$$\bar{\mathbf{x}} = \begin{pmatrix} 1.17 \\ 2.15 \end{pmatrix}. \tag{2.68}$$

This agrees well with the exact solution to 3 decimal places. Computing the residual exactly gives

$$\mathbf{r} = \begin{pmatrix} .003 \\ -.00417 \end{pmatrix}. \tag{2.69}$$

Given the 3-digit accuracy of our computer, **r** is small compared to $\bar{\mathbf{x}}$.

Computing the **LU** factorization to 3 decimal positions and computing **H** exactly gives

$$\mathbf{H} = \begin{pmatrix} 0 & 0 \\ 0 & -.00158 \end{pmatrix}. \tag{2.70}$$

Again, given the 3-digit accuracy of our computer, **H** is small compared to **A**.

Interchanging rows to keep $|l_{ij}| \le 1$ in an attempt to keep the growth of $a_{ij}^{(k)}$ small is called *partial pivoting*. Based on experience, Gaussian elimination with partial pivoting has proven to be stable. Duff, et. al. [10] report at the time of their work that the best a priori bound for Gaussian elimination with partial pivoting is

$$|h_{ij}| \le 5.01\varepsilon\, n\rho \tag{2.71}$$

where

$$\rho \le 2^{n-1} \max_{i,j} |a_{ij}|. \tag{2.72}$$

This is a very loose bound for large systems of equations. Consider $n > 100$ and $\varepsilon \approx 10^{-16}$.

If row and column interchanges are allowed, then it is possible to control growth in $a_{ij}^{(k)}$ even more by ensuring that

$$\left|a_{kk}^{(k)}\right| \ge \left|a_{ij}^{(k)}\right| \text{ for all } i \ge k,\ j \ge k. \tag{2.73}$$

Thus, when $a_{ij}^{(k+1)} = a_{ij}^{(k)} - a_{kj}^{(k)} a_{ik}^{(k)} / a_{kk}^{(k)}$ is computed, we are ensuring the minimum growth possible. This technique is called *full pivoting*. In practice, however, it is not practical due to the large number of comparisons that must be made to determine the pivot.

With sparse systems of equations, partial pivoting may be more restrictive than desired. One of the goals of factoring a sparse system of equations is generating sparse **LU** factors in an attempt to minimize the amount of computation and storage required. The row that satisfies the partial pivoting criterion may have

many non-zero entries. These non-zero entries may cause zero entries in lower rows to become non-zero, thus reducing the sparsity of the factors to a greater degree than another pivot row would. For this reason, the pivot criterion for sparse matrices is generally

$$\left| a_{kk}^{(k)} \right| \geq u \left| a_{ik}^{(k)} \right| \text{ for } i > k \tag{2.74}$$

where

$$0 < u \leq 1 . \tag{2.75}$$

When u=1, this is simply partial pivoting. When u<1, there may be multiple rows from which the pivot row is chosen, each of which should allow only moderate growth in $a_{ij}^{(k)}$. Duff, et. al. give equation (2.71) as an upper bound for growth with

$$\rho \leq \left(1 + u^{-1}\right)^{n-1} \max_{ij} \left| a_{ij} \right| . \tag{2.76}$$

Since there are no useful formulas that indicate the stability of Gaussian elimination and **LU** factorization in practice, the common approach to ensuring the calculations are stable is to measure the precision of the solution after it has been calculated. Before discussing this topic, however, let's explore the concept of matrix and vector sizes.

2.8. Vector and Matrix Norms

The norm of a vector, designated $\|\mathbf{x}\|$, is a measure of the size of the vector. Be definition, a vector norm is a non-negative number that satisfies the following conditions:

$$\|\mathbf{x}\| = 0 \text{ if and only if } \mathbf{x} = \mathbf{0} , \tag{2.77}$$

$$\|\alpha\, \mathbf{x}\| = |\alpha|\, \|\mathbf{x}\| \text{ for any scalar } \alpha , \tag{2.78}$$

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\| \text{ for any vectors } \mathbf{x} \text{ and } \mathbf{y}. \tag{2.79}$$

By definition the p-norm of **x**, denoted $\|\mathbf{x}\|_p$, is given by

$$\|\mathbf{x}\|_p = \sqrt[p]{\sum_i |x_i|^p} . \tag{2.80}$$

Commonly used p-norms are $p = 1,2,\infty$ for which

$$\|\mathbf{x}\|_1 = \sum_i |x_i| \tag{2.81}$$

$$\|\mathbf{x}\|_2 = \sqrt{\sum_i x_i^2} \tag{2.82}$$

$$\|\mathbf{x}\|_\infty = \max_i |x_i| . \tag{2.83}$$

The following relations exist between these norms. n is the dimension of the vector $\mathbf{x}$.

$$0 \le n^{-\frac{1}{2}} \|\mathbf{x}\|_2 \le \|\mathbf{x}\|_\infty \le \|\mathbf{x}\|_2 \le \|\mathbf{x}\|_1 \le n^{\frac{1}{2}} \|\mathbf{x}\|_1 \tag{2.84}$$

A matrix norm $\|\mathbf{A}\|$ is defined a little differently. A vector $\mathbf{x}$ that is multiplied by a matrix $\mathbf{A}$ is transformed to another vector $\mathbf{Ax}$. This leads to the definition

$$\|\mathbf{A}\| = \max_{\|x\|=1} \|\mathbf{Ax}\| . \tag{2.85}$$

For the infinity norm of $\mathbf{A}$ we have

$$\|\mathbf{Ax}\|_\infty = \max_i \left| \sum_j a_{ij} x_j \right| . \tag{2.86}$$

If k is the value of i that maximizes $\|\mathbf{Ax}\|_\infty$ and $|x_j| \le 1$ for all j, then $\|\mathbf{Ax}\|_\infty$ is maximized when $x_j = \text{sign}(a_{kj})$. This gives

$$\|\mathbf{A}\|_\infty = \max_i \sum_j |a_{ij}| . \tag{2.87}$$

In other words, $\|\mathbf{A}\|_\infty$ is the sum of the absolute values of the entries from the row that has the largest such sum. The infinity norm of a matrix is also known as the row norm.

The 1-norm of $\mathbf{A}$ with $\|\mathbf{x}\|_1 = 1$ is also easily computed. By definition of the 1-norm

$$\|\mathbf{Ax}\|_1 = \sum_i \left| \sum_j a_{ij} x_j \right| \tag{2.88}$$

Moving the absolute value inside the summation gives

$$\|\mathbf{Ax}\|_1 \le \sum_i \sum_j |a_{ij}||x_j| \tag{2.89}$$

and changing the order of summation gives

$$\|\mathbf{Ax}\|_1 \le \sum_j \left(\sum_i |a_{ij}| \right) |x_j| \tag{2.90}$$

For a fixed value of j, the inner summation is the sum of the absolute values of the entries of $\mathbf{A}$ in column j. The outer summation is a weighted sum of the column sums. The weights, x_j, are subject to the constraint $\sum_j |x_j| = 1$. The outer summation is thus maximized when $x_j = 1$ for j equal to the column number of $\mathbf{A}$ with the largest sum. Thus

$$\|\mathbf{A}\|_1 = \max_j \sum_i |a_{ij}| . \tag{2.91}$$

The 1-norm of a matrix is also known as the column norm.

The 2-norm of a matrix is the square root of the maximum eigenvalue of $\mathbf{A}^{\mathbf{T}}\mathbf{A}$. The 2-norm is not used in this work and is not discussed further.

Matrix norms have the following properties.

$$\|\mathbf{A}\| = 0 \text{ if and only if } \mathbf{A} = \mathbf{0} \tag{2.92}$$

$$\|\alpha\, \mathbf{A}\| = |\alpha| \|\mathbf{A}\| \text{ for any scalar } \alpha \tag{2.93}$$

$$\|\mathbf{A} + \mathbf{B}\| \le \|\mathbf{A}\| + \|\mathbf{B}\| \tag{2.94}$$

$$\|\mathbf{Ax}\| \le \|\mathbf{A}\| \|\mathbf{x}\| \tag{2.95}$$

$$\|\mathbf{AB}\| \le \|\mathbf{A}\| \|\mathbf{B}\| \tag{2.96}$$

$$0 \le n^{-1} \|\mathbf{A}\|_2 \le n^{-\frac{1}{2}} \|\mathbf{A}\|_\infty \le \|\mathbf{A}\|_2 \le n^{\frac{1}{2}} \|\mathbf{A}\|_1 \le n \|\mathbf{A}\|_2 \tag{2.97}$$

2.9. Checking the Stability of the Calculations

Two forms of stability checking have already been suggested. One, we can compute

$$\mathbf{H} = \bar{\mathbf{L}}\bar{\mathbf{U}} - \mathbf{A} \tag{2.98}$$

to assess the stability of the factorization. Two, we can compute

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\bar{\mathbf{x}} \tag{2.99}$$

to assess the stability of the solution. In practice, computing $\mathbf{H}$ is $O(n^3)$ work, whereas computing $\mathbf{r}$ is $O(n^2)$ work, so examining $\mathbf{r}$ is less effort. Duff, et. al. [10] make the following claims.

1. If $\|\mathbf{r}\|$ is small compared to $\|\mathbf{b}\|$, $\|\mathbf{A}\bar{\mathbf{x}}\|$, or $\|\mathbf{A}\|\|\bar{\mathbf{x}}\|$, then we have done a good job solving the equations.
2. If we have done a good job solving the equations, then $\|\mathbf{r}\|$ is small compared to $\|\mathbf{A}\|\|\bar{\mathbf{x}}\|$.

With regard to the first statement,

1. Duff, et. al. argue that if $\|\mathbf{r}\| << \|\mathbf{A}\|\|\bar{\mathbf{x}}\|$, then $\bar{\mathbf{x}}$ is the exact solution of $(\mathbf{A}+\mathbf{H})\bar{\mathbf{x}} = \mathbf{b}$ where $\|\mathbf{H}\| << \|\mathbf{A}\|$. Thus $\bar{\mathbf{x}}$ is the exact solution of a nearby problem.
2. If $\|\mathbf{r}\| << \|\mathbf{A}\bar{\mathbf{x}}\| \le \|\mathbf{A}\|\|\bar{\mathbf{x}}\|$, then $\bar{\mathbf{x}}$ is again the exact solution of a nearby problem.
3. If $\|\mathbf{r}\| << \|\mathbf{b}\|$, then $\mathbf{A}\bar{\mathbf{x}} = \mathbf{b} - \mathbf{r}$ is a nearby problem and $\bar{\mathbf{x}}$ is the exact solution.

With regard to the second statement,

1. Duff, et. al. also argue that if $\|\mathbf{H}\| << \|\mathbf{A}\|$, then $\|\mathbf{r}\| << \|\mathbf{A}\|\|\bar{\mathbf{x}}\|$.
2. A good solution does not ensure that $\|\mathbf{r}\| << \|\mathbf{b}\|$.

To illustrate this last point, Duff, et. al. give the following example.

$$\begin{pmatrix} 0.287 & 0.512 \\ 0.181 & 0.322 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -0.232 \\ 0.358 \end{pmatrix} \tag{2.100}$$

has the exact solution $x_1 = 1000$, $x_2 = -561$. However, the approximate solution $x_1 = 1000$, $x_{2=} -560$, which would be accurate on a 3-digit computer, has residual

$$\mathbf{r} = \begin{pmatrix} -0.512 \\ -0.322 \end{pmatrix}, \tag{2.101}$$

and $\|\mathbf{r}\|$ is not small compared to $\|\mathbf{b}\|$.

They also point out that the residual does not indicate the behavior of the factorization for other values of **b**. For the case of equation (2.63) with

$$\mathbf{b} = \begin{pmatrix} 0.001 \\ 1.00 \end{pmatrix} \tag{2.102}$$

Gaussian elimination without partial pivoting produces the exact answer $x_1 = 1.00$, $x_2 = 0.00$ on a 3-digit computer even though the calculations are unstable for other values of **b**. For this reason, $\|\mathbf{H}\| = \|\overline{\mathbf{L}}\overline{\mathbf{U}} - \overline{\mathbf{A}}\|$ must be computed to determine the quality of the factorization.

The stability of forward and backward substitution must also be considered when considering the stability of the process. Duff, et. al. argue that solving the triangular system $\mathbf{Tx} = \mathbf{b}$ with calculation error and obtaining the solution $\overline{\mathbf{x}}$ is equivalent to solving $(\mathbf{T} + \mathbf{E})\overline{\mathbf{x}} = \mathbf{b}$ exactly. The bound on **E** is

$$\left| e_{ij} \right| \le (n+1)\psi\varepsilon \left| t_{ij} \right| \tag{2.103}$$

where ψ is a constant and ε is the relative precision of the computer. From this result, they conclude the forward substitution and backward substitution are stable.

Norms tend to measure the large values in a vector or matrix. Stability checks based on norms can be misleading when the entries of a row or column of **A** are much larger than the entries in another row or

column. Small values of a_{ij} can have relatively large corresponding values for h_{ij} even though $\|\mathbf{r}\|$ and $\|\mathbf{H}\|$ are relatively small. Another approach given by Demmel, et. al. [9] is to compute the component-wise relative backward, *BERR*, error given by

$$BERR \equiv \max_i |r_i| / s_i \tag{2.104}$$

where r_i are the components of the residual and

$$s_i = \sum_j |a_{ij}||\overline{x_j}| + |b_i| . \tag{2.105}$$

With component-wise backward error $\bar{\mathbf{x}}$ is the exact solution of the slightly perturbed system $(\mathbf{A} + \mathbf{H})\bar{\mathbf{x}} = (\mathbf{b} + \mathbf{f})$ where

$$|h_{ij}| \le BERR \cdot |a_{ij}| , \tag{2.106}$$

$$|f_i| \le BERR \cdot |b_i| . \tag{2.107}$$

2.10. Ill-Conditioned Problems

Knowing that the calculations have been stable and the extent to which the system of equations must be perturbed in order for $\bar{\mathbf{x}}$ to be an exact solution does not yet answer how accurately $\bar{\mathbf{x}}$ represents the solution of the original problem. If small perturbations of the problem result in large changes to the solution, then $\bar{\mathbf{x}}$ may be an inaccurate estimation of the solution. To illustrate this fact, consider the following simple example taken from Duff, et. al.

$$\begin{pmatrix} 0.287 & 0.512 \\ 0.181 & 0.322 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.799147 \\ 0.502841 \end{pmatrix} . \tag{2.108}$$

The exact solution is

$$\mathbf{x} = \begin{pmatrix} 0.501 \\ 1.28 \end{pmatrix} . \tag{2.109}$$

On a 3-digit computer, **b** must be rounded. When this is done and **A** is factored, we obtain

$$\begin{pmatrix} 1 & \\ 0.631 & 1 \end{pmatrix}\begin{pmatrix} 0.287 & 0.512 \\ & -0.001 \end{pmatrix}\begin{pmatrix} \overline{x_1} \\ \overline{x_2} \end{pmatrix} = \begin{pmatrix} 0.799 \\ 0.503 \end{pmatrix} \tag{2.110}$$

Using forward and backward substitution to solve for $\bar{\mathbf{x}}$ we obtain

$$\bar{\mathbf{x}} = \begin{pmatrix} 1.00 \\ 1.00 \end{pmatrix}. \tag{2.111}$$

Computing **r**, **H**, and *BERR*, we get

$$\mathbf{r} = \begin{pmatrix} 0.000 \\ 0.000 \end{pmatrix}, \tag{2.112}$$

$$\mathbf{H} = \begin{pmatrix} 0.000 & 0.000 \\ 0.000 & 0.000 \end{pmatrix}, \tag{2.113}$$

$$BERR = 0.000\,. \tag{2.114}$$

In fact, if we substitute $\bar{\mathbf{x}}$ into the original set of equations and compute **r** without any rounding error, we obtain

$$\mathbf{r} = \begin{pmatrix} 0.000147 \\ 0.000159 \end{pmatrix}. \tag{2.115}$$

$\|\mathbf{r}\|_\infty$ is still zero to 3 decimal places. $\|\mathbf{b} - \bar{\mathbf{b}}\|_\infty / \|\bar{\mathbf{b}}\|_\infty = 0.002$. Yet $\|\mathbf{x} - \bar{\mathbf{x}}\|_\infty / \|\bar{\mathbf{x}}\|_\infty = 0.499$. The problem is not that the calculations were unstable. The problem is not that the perturbations to the system of equations were large. The problem is that the system of equations is *ill-conditioned* and small perturbations have resulted in large changes to the solution.

Duff, et. al. give the following discussion of ill conditioning. Suppose that **A** is a matrix and **v** and **w** are two vectors such that

$$\|\mathbf{v}\| = \|\mathbf{w}\|, \tag{2.116}$$

$$\|\mathbf{Av}\| >> \|\mathbf{Aw}\|. \tag{2.117}$$

If **b** has the value **Av**, then $\mathbf{Ax} = \mathbf{b}$ has the solution $\mathbf{x} = \mathbf{v}$. If **Aw**, which is small compared to **b**, is added to **b**, then the solution becomes $\mathbf{x} = \mathbf{v} + \mathbf{w}$. But **w** is not small compared to **v**. Thus a small change to **b**

has produced a large change in $\mathbf{x}$. The problem is ill-conditioned. How badly the problem is ill-conditioned is determined by how large the ratio $\|\mathbf{Av}\|/\|\mathbf{Aw}\|$ can become. We can write this ratio as

$$\frac{\|\mathbf{Av}\|}{\|\mathbf{Aw}\|}\frac{\|\mathbf{w}\|}{\|\mathbf{v}\|} = \frac{\|\mathbf{Av}\|}{\|\mathbf{v}\|}\frac{\|\mathbf{A}^{-1}\mathbf{y}\|}{\|\mathbf{y}\|} \le \|\mathbf{A}\|\|\mathbf{A}^{-1}\| \tag{2.118}$$

where $\mathbf{w} = \mathbf{A}^{-1}\mathbf{y}$. The *condition number* of $\mathbf{A}$ is defined to be

$$\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|. \tag{2.119}$$

Duff, et. al go on to consider the variation in $\mathbf{x}$ due to perturbations of the system of equations. Starting with

$$\mathbf{Ax} = \mathbf{b} \tag{2.120}$$

and perturbing $\mathbf{b}$, we can write

$$\mathbf{A}(\mathbf{x} + \delta\ \mathbf{x}) = \mathbf{b} + \delta\ \mathbf{b}\ . \tag{2.121}$$

Subtracting equation (2.120) from (2.121) gives

$$\delta\,\mathbf{x} = \mathbf{A}^{-1}\delta\ \mathbf{b}\ . \tag{2.122}$$

Taking the norms of (2.120) and (2.122) gives

$$\|\mathbf{A}\|\|\mathbf{x}\| \ge \|\ \mathbf{b}\|, \tag{2.123}$$

$$\|\delta\,\mathbf{x}\| \le \|\mathbf{A}^{-1}\|\|\delta\ \mathbf{b}\|. \tag{2.124}$$

Dividing (124) by (123) gives

$$\frac{\|\delta\,\mathbf{x}\|}{\|\mathbf{x}\|} \le \|\mathbf{A}\|\|\mathbf{A}^{-1}\|\frac{\|\delta\ \mathbf{b}\|}{\|\mathbf{b}\|}\ . \tag{2.125}$$

The relative uncertainty in the norm of $\mathbf{x}$ is bounded by the condition number of $\mathbf{A}$ times the relative change in the norm of the perturbed $\mathbf{b}$. The bound given by equation (2.123) may be quite loose, so the bound given by equation (2.125) may also be loose. To see that the bound given by equation (2.123) may be loose, consider the following. If $\|\mathbf{w}\| = \|\mathbf{v}\|$, then it may be that $\|\mathbf{Aw}\| >> \|\mathbf{Av}\|$. If $\mathbf{Av} = \mathbf{b}$, then $\|\mathbf{A}\|\|\mathbf{v}\| = \|\mathbf{A}\|\|\mathbf{w}\| \ge \|\mathbf{Aw}\| >> \|\mathbf{b}\|$.

Performing the analysis for a perturbation of **A** gives what Duff, et. al claim to be a tighter bound on the norm of **x**. Begin by writing

$$(\mathbf{A} + \delta\,\mathbf{A})(\mathbf{x} + \delta\,\mathbf{x}) = \mathbf{b}\,. \tag{2.126}$$

Performing the multiplication, subtracting equation (2.120), and rearranging terms produces

$$\mathbf{A}\delta\,\mathbf{x} = -\delta\,\mathbf{A}(\mathbf{x} + \delta\,\mathbf{x}). \tag{2.127}$$

Multiplying each side by $\mathbf{A}^{-1}$ and taking the norm of each side produces

$$\|\delta\,\mathbf{x}\| \le \left\|\mathbf{A}^{-1}\right\| \|\delta\,\mathbf{A}\| \|\,\mathbf{x} + \delta\,\mathbf{x}\|\,. \tag{2.128}$$

Finally, rearranging terms gives

$$\frac{\|\delta\,\mathbf{x}\|}{\|\,\mathbf{x} + \delta\,\mathbf{x}\|} \le \left\|\mathbf{A}^{-1}\right\| \|\mathbf{A}\| \frac{\|\delta\,\mathbf{A}\|}{\|\mathbf{A}\|}\,. \tag{2.129}$$

The relative uncertainty in the norm of **x** is bound by the condition number of **A** times the relative change in the norm of the perturbed **A**. Since we are bounding relative norms, what we are really bounding are the larger entries of **x**. These bounds do not apply to entries of **x** that are much smaller than the largest entries. The relative error in the norm of **A** can be no less than ε, the relative precision of the computer. This bounds the relative uncertainty in the norm of **x** to be as high as $\varepsilon\,\kappa(\mathbf{A})$.

While $\kappa(\mathbf{A})$ can be computed directly by computing $\mathbf{A}^{-1}$ and forming the product of the norms, the cost of computing $\mathbf{A}^{-1}$ exceeds the cost of solving $\mathbf{Ax} = \mathbf{b}$. Methods are available for estimating $\kappa(\mathbf{A})$ cost effectively, and both of the software packages used in this project do so. Discussing these methods is beyond the scope of this work.

2.11. Scaling

If **A** is such that $\min_{i,j}(a_{ij}) << \max_{ij}(a_{ij})$, then **A** is said to be poorly scaled. There may be individual rows or individual columns that contribute to poor scaling. Suppose $\mathbf{D}_1$ and $\mathbf{D}_2$ are diagonal matrices. The product $\mathbf{D}_1\mathbf{A}$ scales the *i*th row of **A** by $d_{1_{ii}}$. The product $\mathbf{AD}_2$ scales the *j*th column of **A** by $d_{2_{jj}}$. The system of equations $\mathbf{Ax} = \mathbf{b}$ can be scaled as

$$\mathbf{D}_1\mathbf{A}\mathbf{D}_2\mathbf{y} = \mathbf{D}_1\mathbf{b} \tag{2.130}$$

where

$$\mathbf{x} = \mathbf{D}_2\mathbf{y}\ . \tag{2.131}$$

Scaling does not introduce round off error and it can be done relatively quickly since it involves $O(n^2)$ operations. According to Golub and Van Loan [11], a heuristic bound is

$$\frac{\left\|\mathbf{D}_2^{-1}(\bar{\mathbf{x}} - \mathbf{x})\right\|_\infty}{\left\|\mathbf{D}_2^{-1}\mathbf{x}\right\|_\infty} = \frac{\left\|\bar{\mathbf{y}} - \mathbf{y}\right\|_\infty}{\left\|\mathbf{y}\right\|_\infty} \le \mu\,\kappa_\infty(\mathbf{D}_1\mathbf{A}\mathbf{D}_2) \tag{2.132}$$

where μ is the relative error in the norm of $\mathbf{A}$. If $\kappa_\infty(\mathbf{D}_1\mathbf{A}\mathbf{D}_2) << \kappa_\infty(\mathbf{A})$, then we should expect increased accuracy in the relative norm of $\mathbf{D}_2^{-1}\mathbf{x}$. This is the goal of scaling. However, minimizing $\kappa_\infty(\mathbf{D}_1\mathbf{A}\mathbf{D}_2)$ is a difficult mathematical problem. *Simple row scaling* is the scaling of $\mathbf{A}$ such that each row of $\mathbf{D}_1\mathbf{A}$ has approximately the same infinity norm. Row scaling reduces the chances of losing accuracy as a result of adding large number to small numbers in the elimination process. *Row-column equilibration* is the process of choosing and applying $\mathbf{D}_1$ and $\mathbf{D}_2$ such the infinity norm of each row and column of $\mathbf{D}_1\mathbf{A}\mathbf{D}_2$ is within the interval [1/2, 1]. Golub and Van Loan point out that row scaling and row-column equilibration can improve or worsen the accuracy of $\bar{\mathbf{x}}$, and the results of scaling need to be examined for each problem. As an example of the good that simple row scaling can produce, Golub and Van Loan give the following example from Forsythe and Moler. Let

$$\begin{pmatrix} 10 & 100{,}000 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 100{,}000 \\ 2 \end{pmatrix}. \tag{2.133}$$

Row scaling gives

$$\begin{pmatrix} .0001 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}. \tag{2.134}$$

On a 3-digit computer, the solutions for the original problem and the scaled problem are respectively

$$\bar{\mathbf{x}} = \begin{pmatrix} 0.00 \\ 1.00 \end{pmatrix} \text{ and } \bar{\mathbf{x}} = \begin{pmatrix} 1.00 \\ 1.00 \end{pmatrix}. \tag{2.135}$$

The solution of the scaled problem agrees favorable with the exact solution

$$\mathbf{x} = \begin{pmatrix} 1.0001... \\ 0.9999... \end{pmatrix}. \tag{2.136}$$

2.12. Iterative Refinement

Let $\bar{\mathbf{x}}$ be the computed solution of $\mathbf{Ax} = \mathbf{b}$. The residual is given by $\mathbf{r} = \mathbf{b} - \mathbf{A}\bar{\mathbf{x}}$. We may compute a correction to $\bar{\mathbf{x}}$ by solving $\mathbf{Az} = \mathbf{r}$ and obtaining $\bar{\mathbf{z}}$. The refined value for $\mathbf{x}$ is then given by $\bar{\bar{\mathbf{x}}} = \bar{\mathbf{x}} + \bar{\mathbf{z}}$. In fact, this process can be repeated until there is no further improvement in $\mathbf{r}$. Assuming $\mathbf{A}$ has been factored as $\mathbf{LU}$, each iteration involves the computation of $\mathbf{r}$ and the use of forward and backward substitution to solve for $\mathbf{z}$. Not only can iterative refinement improve the accuracy of the computed solution, but $\bar{\mathbf{z}}$ is also an indication of the likely error in $\bar{\mathbf{x}}$. Duff, et. al. point out that this may be the most important role of iterative refinement.

Iterative refinement can also be used to compute changes that occur in $\mathbf{x}$ when small changes are made to $\mathbf{A}$ and $\mathbf{b}$. First, compute $\bar{\mathbf{x}}$ using the unperturbed values of $\mathbf{A}$ and $\mathbf{b}$. Then compute $\mathbf{r}$ and $\bar{\mathbf{z}}$ using the perturbed values. Duff, et. al. tribute Erisman and Reid with the suggestion.

Golub and Van Loan claim that the naive floating point implementation of this algorithm does nothing to improve the accuracy of the computed solution. They do cite work by Skeel whose analysis has shown that $\bar{\bar{\mathbf{x}}}$ does show improvement from the standpoint of backwards error. They go on to suggest a process of mixed precision iterative refinement where $\mathbf{r}$ and $\bar{\mathbf{z}}$ are computed with greater precision than $\bar{\mathbf{x}}$ so that the precision of $\bar{\bar{\mathbf{x}}}$ is truly improved.

2.13. Special Cases

A diagonal entry of a matrix is dominant if its absolute value is greater than or equal to the sum of the absolute values of the other entries in its row or column. The matrix is *diagonally dominant* if all its diagonal entries are dominant. This can be expressed as

$$|a_{kk}| \geq \sum_{i \neq k} |a_{ik}| \qquad k = 1, 2, \ldots n \tag{2.137}$$

or

$$|a_{kk}| \geq \sum_{j \neq k} |a_{kj}| \qquad k = 1, 2, \ldots n\,. \tag{2.138}$$

Gaussian elimination without pivoting is stable for a diagonally dominant matrix. Golub and Van Loan demonstrate that a matrix that is diagonally dominant by columns remains diagonally dominant by columns as **LU** factorization proceeds. Specifically

$$\left|a_{kk}^{(l)}\right| \geq \sum_{i \neq k} \left|a_{ik}^{(l)}\right| \qquad k = 1, 2, \ldots n - l + 1\,. \tag{2.139}$$

This means that no partial pivoting is required.

A similar demonstration can be made for matrices that are diagonally dominant by rows. Write **A** as

$$\mathbf{A} = \begin{pmatrix} \alpha & \mathbf{w}^{\mathbf{T}} \\ \mathbf{v} & \mathbf{C} \end{pmatrix} \tag{2.140}$$

where $\alpha = a_{11}$. After the first step of the factorization we can write

$$\begin{pmatrix} \alpha & \mathbf{w}^{\mathbf{T}} \\ \mathbf{v} & \mathbf{C} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \mathbf{v}/\alpha & \mathbf{I} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \mathbf{C} - \mathbf{v}\mathbf{w}^{\mathbf{T}}/\alpha \end{pmatrix} \begin{pmatrix} \alpha & \mathbf{w}^{\mathbf{T}} \\ 0 & \mathbf{I} \end{pmatrix} \tag{2.141}$$

The outer matrices on the right hand side are the developing **L** and **U** matrices. Let $\mathbf{B} = \mathbf{C} - \mathbf{v}\mathbf{w}^{\mathbf{T}}/\alpha$. **B** is the portion of **A** that remains to be factored. We can show that **B** remains diagonally dominant by rows as follows.

$$\sum_{\substack{j=1 \\ j \neq i}}^{n-1} |b_{ij}| = \sum_{\substack{j=1 \\ j \neq i}}^{n-1} |c_{ij} - v_i w_j / \alpha| \leq \sum_{\substack{j=1 \\ j \neq 1}}^{n-1} |c_{ij}| + \frac{|v_i|}{|\alpha|} \sum_{\substack{j=1 \\ j \neq i}}^{n-1} w_j \tag{2.142}$$

Because **A** is diagonally dominant by rows, we can write

$$\sum_{\substack{j=1 \\ j\neq i}}^{n-1} |c_{ij}| \leq |c_{jj}| - |v_i| \tag{2.143}$$

and

$$\sum_{\substack{j=1 \\ j\neq i}}^{n-1} w_j \leq |\alpha| - |w_i| \tag{2.144}$$

giving

$$\sum_{\substack{j=1 \\ j\neq i}}^{n-1} |b_{ij}| \leq |c_{ij}| - |v_i||w_i|/|\alpha| \leq |b_{jj}|. \tag{2.145}$$

Diagonal dominance by rows limits the growth of $a_{ij}^{(k)}$, contributing to stability. Duff, et. al. show that overall growth in Gaussian elimination for diagonally dominant matrices is limited by

$$\rho = \max_{i,j,k} \left| a_{ij}^{(k)} \right| \leq 2 \max_{ij} |a_{ij}|. \tag{2.146}$$

Another special case is positive definite symmetric matrices. A real, symmetric matrix is *positive definite* if for all non-zero vectors **x** of length n

$$\mathbf{x}^{\mathbf{T}}\mathbf{A}\mathbf{x} > 0. \tag{2.147}$$

Duff, et. al. claim that these matrices arise often in applications because the form $\mathbf{x}^{\mathbf{T}}\mathbf{A}\mathbf{x}$ may represent a non-negative quantity such as energy. Duff, et. al. credit Wilkinson with having shown that Gaussian elimination with diagonal pivots applied to a symmetric positive definite matrix is stable in the sense that overall growth of $a_{ij}^{(k)}$ is limited by

$$\rho = \max_{i,j,k} \left| a_{ij}^{(k)} \right| \leq \max_{ij} |a_{ij}|. \tag{2.148}$$

According to Lay [12], a symmetric positive definite matrix can be factored as

$$\mathbf{A} = \mathbf{LDL}^{\mathbf{T}} \tag{2.149}$$

where **D** is a diagonal matrix; that is **D** has non-zero entries only on the diagonal. Taking $\mathbf{U} = \mathbf{DL}^{\mathbf{T}}$, the factorization of **A** can be done with approximately half the number of arithmetic operations as a full **LU** factorization of **A**. Such a factorization is called a Choleski factorization.

3. Issues Regarding Sparse Systems of Linear Equations

In addition to the mathematical issues regarding systems of linear equations presented in the previous chapter, there are issues specific to sparse systems of linear equations. Some of these issues are mathematical, while others, such as data storage schemes, are computer related. There is no exact definition of a sparse system. Matrices that have many more zero entries than non-zero entries characterize sparse systems. The systems of equations generated by FEM and the ice sheet model meet this requirement. The ice sheet equations have on the order of ten to one hundred non-zero entries per row, but there are thousands or tens of thousands of variables per row.

In addition to getting an acceptable solution, two goals of solving sparse systems of equations on a computer are minimizing the storage used and minimizing the execution time needed. Since the arithmetic operations are generally performed by the computer's floating point processor, arithmetic operations are measured in floating point operations, or FLOPs. A measure of the speed of the computer is floating point operations per second, or FLOPS. Matrix and vector values must be stored in some sort of real number format. Typically, a single precision or double precision floating point representation is used. A single precision floating point number requires 4 bytes of memory and gives approximately 6 decimal digits of precision, while a double precision floating point number requires 8 bytes of memory and gives approximately 16 decimal digits of precision. A system of 24,000 equations represented by double precision floating point numbers is not uncommon for ice sheet modeling. If the system is represented as an array of 24,000 by 24,000 entries, approximately 4.6 gigabytes of storage are required to store **A** and solving the system requires approximately 10^{13} FLOPs. For a typical standalone processor that can perform 10^9 floating point operations per second, 10^4 seconds or 2 to 3 hours of processing time are required. This time estimate assumes **A** will fit in main storage. If it doesn't, the execution time may be much longer. Yet **A** contains only 1.9 million non-zero entries! Also, keep in mind that ten thousand systems of equations will need to be solved to model the evolution of a glacier over an ice age. Therefore we must take advantage of the sparsity of the equations to reduce execution time and storage overhead.

3.1. Data Storage Schemes

While representing the matrix **A** by a two dimensional array is not storage efficient for sparse systems of equations, it nonetheless has desirable attributes. So long as the array is small enough to fit in main storage, any element of **A** is directly addressable and can be retrieved quickly. Also, if a zero entry should become non-zero in the course of solving the system, memory has already been allocated for the entry, so there are no complicated memory allocation considerations. Entries can be found and new entries can be added with cost O(1). Also, there is no additional overhead for storing row and column information. This information is an intrinsic part of the addressing scheme.

Triplet notation is another representation for **A**. In this form only the non-zero entries are stored. For each non-zero entry we store the row number, column number, and the value of the entry. The row and column numbers can be stored as two byte integers for systems of equations with less than 32,767 variables, or as four byte integers for systems with more variables. The entry values must be stored in a suitable real number format. We will assume that entry values are stored as double precision real numbers from this point forward. The components of the triplet can be stored in three one-dimensional arrays or as a structure of two integers and a double precision real. Using three one-dimensional arrays works well with Fortran 77 because the language does not have support for more complex data structures. The ice sheet model is in fact written in Fortran 77. Figure 3.1 illustrates triplet storage using arrays.

Count	4					
Row	1	1	2	3	-	...
Column	1	2	3	4	-	...
Value	2.5	10.1	5.3	-10.1	-	...

Figure 3. 1. Triplet storage using arrays.

The three components of the triplet share the same index value so ROW(I), COLUMN(I), and VALUE(I) represent the triplet of the i'th entry. An additional integer variable stores the number of entries. Entries may be ordered by row and column as in this example, or they may be unordered. If data is ordered, a binary search can be used to locate specific entries with cost O(log2(n)). In this example the search key is a segmented key consisting of row and column number. If the data is unordered, then a sequential search with cost O(n) is required to find an entry. If a new entry is added to an unordered set of entries, it can simply be added as the N+1 entry with cost O(1). If the data is ordered, then all entries that follow the new entry must be shifted up one position in the arrays to make room for the new entry. In this case the insertion operation has cost O(n) where n is the number of entries already stored. O(log2(n)) work is required to determine where to insert the new item, and cost O(n) work is required to move the entries that are ordered after the new one to make room for it. Two options exist for deleting an entry. The value of the entry can be set to zero, or the entry can be physically removed from the arrays. If the value is set to zero, then the cost is simply the cost of the search plus the cost to store the zero. If the entry is physically removed, then the cost is the cost of the search plus the cost of moving all entries above the removed entry down by one.

Languages such as C and C++ have language support for structures consisting of a collection of intrinsic data types. The structures may be stored in an array or a linked list. If they are stored in an array, then the same issues of ordered versus unordered data apply and searching, adding, and deleting have the same costs as the implementation using three separate arrays.

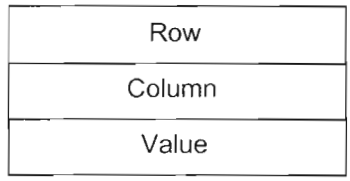

Figure 3. 2. Structure for storing a triplet.

If structures are stored in a linked list, the list may be singly linked or doubly linked. For singly linked lists, each structure has a pointer to the next structure in the list. The pointer of the last item in the list is

given a null value to indicate the end of the list. One additional pointer variable is used to point to the first entry in the list.

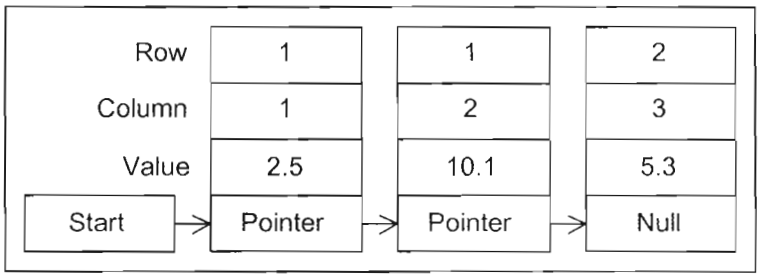

Figure 3. 3. Singly linked list of triplets.

Singly linked lists are typically unordered. The only way to find an entry in a linked list is to search the list an entry at a time beginning with the first entry in the list. Thus finding an entry has cost O(n). If a linked list is unordered, then searching for an item that is not in the list will cause every entry in the list to be checked. If a linked list is ordered, then searching for an item that is not in the list can be terminated as soon as we have encountered an entry that follows the one for which we are searching. Still, however, the cost of searching is O(n). An entry can be added to an unordered list in time O(1). The new entry is simply added to the beginning of the list. The starting pointer is updated to point to the new entry and the pointer of the new entry points to the entry that was first. If the list is ordered, then adding a new entry means searching for the place where the entry belongs, changing the pointer of the preceding entry to point to the new entry, and setting the pointer of the new entry to point to the following entry. The cost of adding an entry to an ordered list is O(n) because of the search. Deleting an entry always involves searching for the entry, so deleting an entry also has cost O(n). Once the entry to be deleted has been found, the value can be set to zero or the entry can be physically removed. To remove the entry, the pointer of the preceding entry is changed to point to the entry following the entry being deleted. Memory for the deleted entry can be deallocated.

For a doubly linked list each structure has two pointers: one to the next entry in the list and another to the previous entry. This allows the list to be traversed forwards and backwards. Two additional pointer variables are used to point to the first and last entries in the list.

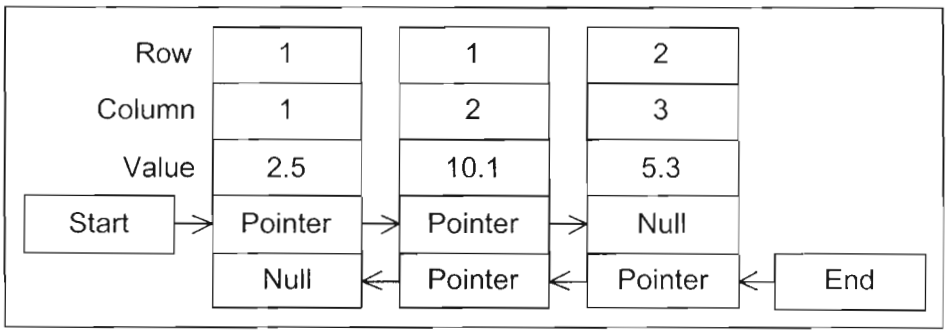

Figure 3. 4. Doubly linked list of triplets.

Doubly linked lists are normally maintained as ordered lists. If a sequence of operations involves entries that are nearby one another, the links can be followed in the appropriate direction to find the entries with a cost that is much less than searching from the beginning of the list. The same principal applies when entries are added or deleted. If we have no knowledge of a nearby entry, then searching, adding, and deleting have cost O(n) like a singly linked list. In addition, the need to maintain two sets of pointers adds a little more complexity to the programming.

While Fortran 77 does not support structures of primitive data types, linked lists can still be implemented in the language using arrays. A pointer field in a structure can be replaced by an array of pointers. Pointer values are the index values of the data they reference in the data arrays. Memory management can be implemented by chaining unused array entries as a singly linked free space chain.

Triplet notation requires storage overhead for row and column information. If the triplets are stored as linked lists, additional storage is required for pointers. Most sparse storage schemes require storage overhead above the storage required for the values. Storage of banded matrices is an exception that will be discussed later.

Triplet notation is particular well suited as an input/output format. It is human readable and a common format for exchanging data.

While the discussion so far has focused on sparse matrices, similar techniques and structures can be used to store sparse vectors. Instead of having row, column, and value attributes, a sparse vector has only column and value attributes.

Another scheme for storing sparse matrices is *compressed column format* as illustrated in the next figure.

N	3					
Column Start	1	3	5	6	-	...
Row	1	2	2	3	1	...
Value	2.5	10.1	5.3	7.8	-10.3	...

Figure 3. 5. Compressed column format.

Values and corresponding row numbers are explicitly stored. The column number is used as an index to access the Column Start array. The values of the Column Start array are the indices for the first Value in each column. In the example above, column 1 has the value 2.5 in row 1 and 10.1 in row 2. Column 2 has the value 5.3 in row 2 and 7.8 in row 3. Finally, Column 3 has the value –10.3 in row 1. N contains the number of columns in the matrix **A**. The number of entries in column j is

$$ColumnStart(j+1) - ColumnStart(j).$$

To keep the scheme consistent, the N+1 element of the Column Start array is set to the number of values stored in Value plus 1. The values are intrinsically ordered by column. In addition, the values within a column are generally ordered by row, producing a total order. When the data are totally ordered, the value at a specific row and column can be found very quickly. Suppose we want to find the value of a_{ij}. The values *ColumnStart(j)* and *ColumnStart(j*+1*)*-1 define the lower and upper bound of the index values of *Row* for column j. Use these bounds to perform a binary search on *Row* for the value i. If i is not found,

then $a_{ij} = 0$. If i is found at say index k, then $a_{ij} = Value(k)$. Thus the cost of searching is O(log2(c)) where c is the number of values in a column.

Adding a new entry is more complicated. First we must determine where the entry belongs in the Value array. Once that position has been determined, all values in Row and Value at that position and above must be moved up one position. The value of the new entry must be stored in the opened space in Value and the row number of the entry must be stored in Row. Finally, all values in Column Start for column numbers greater than the entry just added must be incremented by 1 to reflect the movement of the data in Row and Value. Overall, adding a new entry has cost O(n) where n is the number of non-zero entries in **A.**

As before, deletion of an entry can be accomplished in one of two ways. First, the value of the entry can be set to zero for the cost of doing a search and storing the zero. Second, we can physically remove the entry from the data structure. Removing an entry is essentially the opposite of adding an entry. First we search for the Value index of the entry. Then we move all data above that index in Value and Row down one position. Finally, we update the values in Column Start to reflect the movement of data in Row and Value. Overall, this scheme has cost O(n) where n is again the number of non-zero entries in **A**.

All entries in a specific column can be efficiently retrieved in compressed column format. However, retrieving all entries in a specific row is much less efficient. To do so we must perform a binary search of all N columns. Ordered triplet data has this same weakness depending upon whether the row value or the column value comes first in the segmented key. Compressed row format is an alternative to compressed column format. All entries in a specific row can be efficiently accessed, but we lose efficient access to all entries in a specific column. Figure 3.6 illustrates compressed row.

N	3					
Row Start	1	3	5	6	-	...
Column	1	3	1	2	2	...
Value	2.5	-10.3	10.1	5.3	7.8	...

Figure 3. 6. Compressed row format.

Searching, adding, and deleting entries in compressed row format are the same as compressed column format, except the roles of row and column are interchanged.

The cost of adding and deleting entries in the compressed row and compressed column formats can be mitigated by storing the entries in a row or column as a linked list. Figure 3.7 illustrates storing rows of entries as a linked list.

Row Start	2	3	5	0	0	...
Column	2	1	1	3	2	...
Value	5.3	2.5	10.1	-10.3	7.8	...
Link	0	4	1	0	0	...

Figure 3. 7. Linked row format.

The Row Start array is indexed by row number. The value from the Row Start array is the index value for retrieving the column number and value of the first entry in the row from the Column and Value arrays. The Link value is the index number for the next entry in the row. A link value of zero indicates the end of the list. Searching for a value in a row requires a linear search of the list for that row, so the cost of the search is O(r) where r is the number of entries in a row. If entries within a row are ordered by column

number, the search can be terminated as soon as we encounter a column number larger than the column number we are searching for. If entries within a row are not ordered, then the search can only be terminated if the entry is found or the end of the list is reached. If the list is unordered, then a new entry can be added with cost O(1) by simply adding it to the head of the list. If the list is ordered, then we must search the list for the position of the new entry and adjust the link value of the preceding entry to point to the new one. The overall cost is O(r). Deleting an entry requires searching for it first. If the deleted item is removed, then the link value of the preceding entry must be updated to point to the following entry. Otherwise, the value of the deleted entry can simply be set to zero.

Gaussian elimination and LU factorization require access to elements by row and by column. This can be accomplished by having both row links and column links as illustrated in Figure 3.8.

Row Start	2	3	5	0	0	...
Column Start	2	1	4	0	0	...
Row	2	1	2	1	3	...
Column	2	1	1	3	2	...
Value	5.3	2.5	10.1	-10.3	7.8	...
Row Link	0	4	1	0	0	...
Column Link	5	3	0	0	0	...

Figure 3. 8. Linked row and column format.

The storage for the starting pointers, row and column values, and links is three times the storage for the values if 4-byte integer values and 8-byte double precision values are used. Storage overhead can be traded

for execution overhead by eliminating the row and column arrays and storing the negated values of row numbers and column numbers as the end of list link values as shown in the next figure.

Row Start	2	3	5	0	0	...
Column Start	2	1	4	0	0	...
Value	5.3	2.5	10.1	-10.3	7.8	...
Row Link	-2	4	1	-1	-3	...
Column Link	5	3	-1	-3	-2	...

Figure 3. 9. Linked row and column format with embedded row and column numbers.

When accessing elements by column, the row links for each entry are followed to the end of the row list to determine the row number. Likewise, when accessing elements by row, the column links for each entry are followed to the end of the column list to determine the column number.

3.2. Common Operations On Sparse Matrices and Vectors

As with any data structures problem, the optimal data structure for a given problem depends upon the access requirements of the problem as well as the performance attributes of the data structure. From the viewpoint of the ice sheet model, the operations that will be performed over and over are the computation of the entries of **A** and **b** in the equation **Ax**=**b**. In particular, individual entries throughout **A** and **b** are directly accessed through a series of arithmetic accumulations as the ice sheet model runs. Once an iteration of the model is finished, the entries need to be zeroed and the computational sequence repeated. Specific choices made for the data structures are discussed later.

From the viewpoint of solving systems of linear equations, the arithmetic operations that will be performed repeatedly are

1. The addition of a sparse vector scaled by a constant to another sparse vector (Gaussian row replacement)
2. The inner product of two sparse vectors (the formation of one entry in a matrix-matrix product or a matrix-vector product).

To reap the benefits of sparsity, these operations should have cost that is proportional to the count of non-zero entries in the two vectors. If costs of the operations are proportional to the lengths of the vectors, then the costs of these operations would be of the same order as the costs of working with dense vectors and we would gain little or no efficiency.

3.2.1. Addition of Sparse Vectors

Let's consider forming the sum of two sparse vectors **x** and **y**. Unless the row numbers of the non-zero entries of **x** are a subset of the row numbers of the non-zero entries of **y**, or vice-versa, the row numbers of the entries in **x** + **y** will not be a subset of the row numbers of **x** or **y**. By "non-zero entries" we mean the entries that are explicitly stored in the sparse representation of the vector, even if the value of some of these entries happen to be zero. Thus the addition of sparse vectors generally entails the insertion of new entries.

Of the data structures presented, the ones based on linked lists have the lowest insertion costs. Within the linked list structures, we can choose ordered lists or unordered lists. In addition, we must consider whether the resultant vector overwrites one of the two vectors being added or if it must be stored as a separate vector. For example, in Gaussian elimination the resultant vector replaces one of the vectors being added. Let's consider each of the four possible scenarios.

Case 1: the sparse representation is ordered and the resultant vector replaces one of the existing vectors. The following pseudo code demonstrates how addition can be performed with cost O(c1+c2) where c1 is the number of entries in vector **v1** and c2 is the number of entries in **v2** and **v1** + **v2** overwrites **v1**.

```
ptr1a = 0
ptr1 = v1.StartIndex
ptr2 = v2.StartIndex
While ptr1 <> 0
    While ptr2 <> 0
        If v1(ptr1).Column > v2(ptr2).Column
            Insert v2(ptr2) After v1(ptr1a) Advancing ptr1a
            ptr2 = v2(ptr2).Link
        Else If v1(ptr1).Column = v2(ptr2).Column
            v1(ptr1).Value = v1(ptr1).Value + v2(ptr2).value
            ptr2 = v2(ptr2).Link
            ExitWhile
        Else
            ExitWhile
    EndWhile
    ptr1a = ptr1
    ptr1 = v1(ptr).Link
EndWhile
While ptr2 <> 0
    Insert v2(ptr2) After v1(ptr1a) Advancing ptr1a
    ptr2 = v2(ptr2).Link
EndWhile
```

Figure 3. 10. Addition of ordered sparse vectors with overwriting.

Variables `ptr1` and `ptr2` point to the active entries in `v1` and `v2` respectively. Variable `ptr1a` points to the entry of `v1` that is prior to the active entry. When `ptr1a` is zero, there is no prior entry in `v1`. Any entries in `v2` that have not been processed and are ordered before the active entry of `v1` are added to `v1` by the `Insert x(a) After y(b) Advancing b` statement. This statement represents the processing required to add entry `x(a)` to the linked list `y` after entry `b` and advance `b` to the newly added entry. It has cost O(1) for singly and doubly linked lists. It performs whatever memory management is necessary to add

the new entry. If we are dealing with arrays in Fortran 77, then this would mean using the next available array entries. If the active entries of `v1` and `v2` have the same column number, the value of the active entry of `v2` is added to the value of the active entry of `v1`. Otherwise, `ptr1` and `ptr1a` are advanced to the next entries and the process repeats. Once all entries in `v1` have been processed, any unprocessed entries in `v2` are added to the end of `v1`.

Case 2: the sparse representation is ordered and the resultant vector is stored as a new list. The following pseudo code demonstrates how addition can be performed with cost O(c1+c2) where c1 is the number of entries in vector **v1** and c2 is the number of entries in **v2** and **v1** + **v2** overwrites **v1**.

```
ptr1 = v1.StartIndex
ptr2 = v2.StartIndex
Initialize v3
ptr3a = v3.startIndex
While ptr1 <> 0
    While ptr2 <> 0
        If v1(ptr1).Column > v2(ptr2).Column
            Insert v2(ptr2) After v3(ptr3a) Advancing ptr3a
            ptr2 = v2(ptr2).Link
        Else If v1(ptr1).Column = v2(ptr2).Column
            Insert v1(ptr1)+v2(ptr2) After v3(ptr3a) Advancing ptr3a
            ptr2 = v2(ptr2).Link
            ExitWhile
        Else
            ExitWhile
    EndWhile
    Insert v1(ptr1) after v3(ptr3a) Advancing ptr3a
    ptr1 = v1(ptr).Link
EndWhile
While ptr2 <> 0
    Insert v2(ptr2) After v3(ptr3a) Advancing ptr3a
    ptr2 = v2(ptr2).Link
EndWhile
```

Figure 3. 11. Addition of ordered sparse vectors.

This case is similar to Case 1. The `Initialize v3` statement instantiates a new list by setting the start index of the list to zero. The `v1(ptr1) + v2(ptr2)` operand in the insert statement signifies that the sum of the values of `v1` and `v2` is to be added as the value of the new entry. The column of the new entry can be taken from the column of either `v1` or `v2` since they are equal.

Case 3: the sparse representation is unordered and the resultant vector replaces one of the existing vectors. The following pseudo code demonstrates how addition can again performed with cost O(c1+c2). Assume **v1** is overwritten and w is an array of n real numbers, all initialized to zero, where n is the dimension of **v1** and **v2**.

```
ptr = v2.StartIndex
While ptr <> 0
    w[v2(ptr).Column] = v2(ptr).Value
    ptr = v2(ptr).Link
EndWhile
ptr = v1.StartIndex
While ptr <> 0
    col = v1(ptr).Column
    If w[col] <> 0
        v1(ptr).Column = v1(ptr).Column + w[col]
        w[col] = 0
    Endif
    ptr = v1(ptr).Link
EndWhile
ptr = v2.StartIndex
While ptr <> 0
    col = v2(ptr).Column)
    If w[col] <> 0
        Insert v2(ptr) AtHead v1
        w[col] = 0
    Endif
    ptr = v2(ptr).Link
EndWhile
```

Figure 3. 12. Addition of unordered sparse vectors with overwriting.

This algorithm is a three-step process. In the first step, v2 is copied to w. In the second step, each element of v1 is examined. If there is a non-zero entry in w corresponding to an entry in v1, then the value in w is added to the value in v1 and the value in w is set to zero. In the third step, each element of v2 is examined. If there is a non-zero entry in w corresponding to an entry in v2, then that entry from v2 is inserted at the head of v1 with cost O(1) and the value in w is set to zero. At the end of the process w is still initialized to zero and is ready for another vector addition operation.

Case 4: the sparse representation is unordered and the resultant vector is added as a new vector. This case is similar to Case 3.

```
Initialize v3
ptr = v2.StartIndex
While ptr <> 0
    w[v2(ptr).Column] = v2(ptr).Value
    ptr = v2(ptr).Link
EndWhile
ptr = v1.StartIndex
While ptr <> 0
    col = v1(ptr).Column
    If w[col] <> 0
        Insert v1(ptr) + w[col] AtHead v3
        w[col] = 0
    Else
        Insert v1(ptr) AtHead v3
    Endif
    ptr = v1(ptr).Link
EndWhile
ptr = v2.StartIndex
While ptr <> 0
    col = v2(ptr).Column)
    If w[col] <> 0
        Insert v2(ptr) AtHead v3
        w[col] = 0
    Endif
    ptr = v2(ptr).Link
EndWhile
```

Figure 3. 13. Addition of unordered sparse vectors.

List `v3` is initialized as a new list before a similar three-step process begins. The first step copies `v2` to `w`. The second step examines each entry of `v1`. If an entry has a corresponding value in `w`, then the sum of the values from `v1` and `w` are inserted as a new entry in `v3` and the value in `w` is set to zero. Otherwise the entry from `v1` is inserted into `v3`. The third step examines every entry in `v2`. If the corresponding entry in `w` is not zero, then the entry from `v2` is inserted into `v3` and the value in `w` is set to zero. At the end of the process `w` is still initialized to zero and is ready for another vector addition operation.

3.2.2. Inner Product of Sparse Vectors

The inner product of two sparse vectors is an easier problem than the sum of two sparse products. There are only two classes of problems to consider: are the vector entries ordered or unordered. For ordered entries we can borrow techniques from the ordered vector sum algorithm.

```
ptr1 = v1.StartIndex
ptr2 = v2.StartIndex
product = 0
While ptr1 <> 0
    While ptr2 <> 0
        If v1(ptr1).Column > v2(ptr2).Column
            ptr2 = v2(ptr2).Link
        Else If v1(ptr1).Column = v2(ptr2).Column
            product = product + v1(ptr1).Value + v2(ptr2).Value
            ptr2 = v2(ptr2).Link
            ExitWhile
        Else
            ExitWhile
    EndWhile
    ptr1 = v1(ptr).Link
EndWhile
```

Figure 3. 14. Inner product of ordered sparse vectors.

The first entries of `v1` and `v2` are made the active entries. The variable `product`, which will contain the inner product at the conclusion of the calculations, is initialized to zero. All entries in `v2` that precede the active entry of `v1` are skipped. If the active entries of `v1` and `v2` have matching column numbers, then the product of the values of `v1` and `v2` are accumulated in `product`. Finally, the next entry of `v1` is made active and the process is repeated until all entries in `v1` have been processed.

If the vectors are unordered, then we borrow techniques from the unordered vector sum algorithm. As before, `w` is an array of n real numbers where n is the dimension of vectors **v1** and **v2**. The entries of `w` are presumed to be zero before the algorithm runs.

```
ptr = v2.StartIndex
While ptr <> 0
    w[v2(ptr).Column] = v2(ptr).Value
    ptr = v2(ptr).Link
EndWhile
product = 0
ptr = v1.StartIndex
While ptr <> 0
    product = product + v1(ptr).value * w[v1(ptr).Column]
    ptr = v1(ptr).Link
EndWhile
ptr = v2.StartIndex
While ptr <> 0
    w[v2(ptr).Column] = 0
    ptr = v2(ptr).Link
EndWhile
```

Figure 3. 15. Inner product of unordered sparse vectors.

There are three steps in the process. The first step copies the values of the entries in `v2` to the corresponding entries in `w` by column number. The second step forms the inner product by accumulating the products of the entries of `v1` with the corresponding entries in `w`. The third step resets the entries in `w` corresponding to the entries of `v2` to zero. Thus `w` is initialized to zeros at the end of the process and is ready to be used again.

3.3. Conflicting Optimization Requirements for Data Structures

Various processing requirements can cause conflicts when choosing the optimal data structure for a problem. This is indeed what happens with the ice sheet model. To optimize the FEM portion of the model, access to entries of **A** by row and column must have as little overhead as possible. After the first iteration of the model we know which entries of **A** are non-zero. With this information in hand, the compressed column data structure and the compressed row data structure can provide good performance. However, the question of what data structure to use for the first iteration of the model remains open.

We can expect the process of solving the system of equations to involve sums of sparse vectors. This will generate new vectors with the requirement to insert additional entries. The linked list representations have good performance for the insertion operation, but the compressed column and compressed row data structures have poor performance in this regard.

If we had to choose one data structure, the ordered linked list representation would probably provide the best compromise between accessing entries by row and column and inserting new entries as a result of vector addition. The unordered linked list representation would probably be a close second. However, there is a third possibility. The FEM process is distinct from the equation solving process. Both processes are computationally intensive. The optimum data structure can be chosen for the FEM process, and when the FEM calculations are complete, the FEM data structure can be copied to a data structure that is optimum for solving the system of equations. This is in fact what happens with software packages for solving systems of linear equations. A data structure is specified for passing **A** and **b** to the software package, but the software package uses alternative structures internally for optimum performance. The specific structures used in this work will be discussed later.

3.4. Markowitz Cost: Row and Column Orderings for Optimized LU Factorization

We have already shown how permutation matrices can be used to change the row and column orderings of a matrix. In addition, we have also shown how a system of equations is solved with **LU** factorization when row and column permutations have been performed. The following example illustrates how row and column orderings of a sparse matrix can affect the sparsity of the **L** and **U** factors. Our goal is to minimize the number of non-zero entries in $\mathbf{A}^{(k)}$ and hence minimize the number of non-zero entries in **L** and **U**. We have shown that the amount of work required to add sparse vectors can be as low as O(c1+c2) where c1 and c2 are the number of non-zero entries in each of the vectors. By optimizing sparsity, we minimize the cost of factorization. Suppose **A** has non-zero entries in the positions marked with an x in the following figure. For the moment, also assume that any set of row and column interchanges will maintain stability. This example has been taken from Duff, et. al. [10].

```
x x x x x x x x
x x
x   x
x     x
x       x
x         x
x           x
x             x
x               x
```

Figure 3. 16. Sparsity pattern for matrix **A**.

Using a_{11} as the first pivot will potentially cause all positions of $\mathbf{A}^{(2)}$ to become non-zero. Every row of **A** has a non-zero entry in the first column, so a multiple of the first row will be added to every other row. In addition, the first row of **A** has a non-zero entry in every column, so non-zero entries will be added to every column of every row in **A**.

If we interchange the first and last rows of **A** we obtain the following sparsity pattern.

```
x               x
x x
x   x
x     x
x       x
x         x
x           x
x x x x x x x x
```

Figure 3. 17. Swapping first and last rows of **A**.

Now, using a_{11} as the first pivot will only cause non-zero entries to be added to the last column of $\mathbf{A}^{(2)}$. This sparsity pattern will prevail as each $\mathbf{A}^{(k)}$ is computed, so the number of arithmetic operations is kept low.

If we instead interchange the first and last columns of **A** we obtain the following sparsity pattern.

```
x x x x x x x x
  x             x
    x           x
      x         x
        x       x
          x     x
            x   x
              x x
x               x
```

Figure 3. 18. Swapping first and last columns of **A**.

Now, using a_{11} as the first pivot will only cause non-zero entries to be added to the last row of $\mathbf{A}^{(2)}$. This sparsity pattern will prevail as each $\mathbf{A}^{(k)}$ is computed, so again the number of arithmetic operations is kept low.

As a final case, consider what happens when the first row is interchanged with the last row and the first column is interchanged with the last column.

```
x               x
  x             x
    x           x
      x         x
        x       x
          x     x
            x   x
x x x x x x x x
```

Figure 3. 19. Swapping first and last rows and first and last columns of **A**.

Now the sparsity pattern of **A** will prevail as each $\mathbf{A}^{(k)}$ is computed.

A greedy strategy for choosing pivots to minimize the fill-in of zero entries as the factorization progresses is attributed to Markowitz in 1957 by Duff, et. al. For each potential pivot Markowitz counts the number of non-zero entries in the row of the potential pivot, $r_i^{(k)}$, and the number of non-zero entries in the column of the potential pivot, $c_j^{(k)}$, of the sub-matrix that remains to be factored at stage k. For each row that has a non-zero entry in the column of the potential pivot, as many as $c_j^{(k)} - 1$ non-zero entries may be added to

the unfactored portion of the matrix. There are $r_i^{(k)} - 1$ such rows, so the product $(r_i^{(k)} - 1)(c_j^{(k)} - 1)$ is the maximum number of non-zero entries that could be added if the potential pivot is used as the next pivot. The product $(r_i^{(k)} - 1)(c_j^{(k)} - 1)$ is called the *Markowitz count*. The potential pivot with the lowest Markowitz count is chosen as the next pivot.

For the example matrix in Figure 3.16, the Markowitz algorithm chooses row and column orderings such that the permuted value of **A** will have the sparsity pattern shown in Figure 3.19. The first time a pivot is chosen, there are seven potential pivots with a Markowitz cost of 1, 14 potential pivots with a Markowitz cost of 7, and one potential pivot with a Markowitz cost of 49. In the case of a tie we may choose from any of the lowest cost pivots. If $a_{88}^{(1)}$ is chosen as the pivot, then the first and last rows of **A** and the first and last columns of **A** are immediately interchanged and no additional interchanges will take place. If another potential pivot with a Markowitz cost of 1 is chosen, then there will be a sequence of row and column permutations as the factorization process progresses. When the factorization is complete, the combined permutations will be equivalent to interchanging the first and last rows of **A** and interchanging the first and last columns of **A**.

The number of multiplication and division operations that take place at stage k of the factorization is $(r_i^{(k)} - 1)c_j^{(k)}$. Therefore the Markowitz strategy also tends to minimize the number of arithmetic operations at each stage.

Computing the Markowitz cost of all potential pivots is prohibitively large for large sparse matrices. A modified strategy is to only look at the potential pivots in the first few columns of the remaining unfactored portion of **A**.

The Markowitz algorithm does not consider stability of the calculations. In practice, the relative size of the absolute value of the potential pivot must also be considered. We want to choose a relatively large pivot. Typically, a potential pivot must also satisfy the requirement

$$\left| a_{ij}^{(k)} \right| \geq u \max_{l} \left| a_{lj}^{(k)} \right| \tag{3.1}$$

where

$$0 < u \leq 1 . \tag{3.2}$$

The value u is called the *threshold parameter*. A value such as $u = 0.1$ often works well in practice.

Finally, we must recognize that the Markowitz strategy is a local strategy. We minimize the amount of fill-in at each stage of the factorization with the hope that it will minimize the total fill-in for all stages. However, there is no guarantee that it will. In practice, the Markowitz strategy has worked well.

3.5. Minimum Degree Pivot Selection

If $\mathbf{A}^{(k)}$ is symmetric then $r_l^{(k)} = c_l^{(k)}$ and the Markowitz count of any entry such as $a_{lm}^{(k)}$ is $(r_l^{(k)} - 1)(r_m^{(k)} - 1)$. If $a_{ll}^{(k)}$ is the diagonal entry with minimum Markowitz count, then there is no off-diagonal entry with a lower Markowitz count. If $\mathbf{A}$ is diagonally dominant or positive definite, then diagonal pivots are stable. Finally, if $\mathbf{A}^{(k)}$ is symmetric and we use a diagonal pivot, then $\mathbf{A}^{(k+1)}$ is also symmetric. Thus at each stage of the factorization we simply choose the diagonal pivot $a_{ii}^{(k)}$ corresponding with

$$r_i^{(k)} = \min_{l} r_l^{(k)} . \tag{3.3}$$

This algorithm is called *minimum degree*. Duff, et. al. credit Tinney and Walker with this discovery made in 1967.

The minimum degree pivot algorithm can be easily implemented. We simply maintain an array of the r_i values for all the rows. These values are easily updated as the factorization process proceeds. Whenever a new non-zero entry is added to a row through row replacement, we increment r_i for the row. Whenever a column is eliminated from a row, we reduce r_i for the row. Choosing the next pivot is simply a matter of scanning the array to find the row with the smallest r_i and using the diagonal entry of that row as the next

pivot. Combining this algorithm with the sparse data structures and the algorithms for adding sparse vectors, we can implement a fast **LU** factorization algorithm for sparse, positive definite, symmetric matrices.

If **A** is not symmetric, we must maintain the values of r_i and c_j as the factorization progresses. Choosing a pivot requires us to identify the non-zero pivots, compute the Markowitz count from the values of r_i and c_j, and ensure the count is minimized subject to the stability constraint. Choosing a pivot is significantly more work when **A** is not symmetric. If **A** is nearly symmetric and diagonal pivots are stable, the minimum degree algorithm is often used because of its efficiency.

Minimum degree, and hence the Markowitz strategy, do not necessarily minimize fill-in. Consider a symmetric, positive definite matrix with the following sparsity pattern from Duff, et. al.

```
x x x x
x x x x
x x x x
x x x x x
      x x x
        x x x x x
          x x x x
          x x x x
          x x x x
```

Figure 3. 20. Non-optimal fill-in with minimum degree.

Minimum degree selects the diagonal entry in the fifth row as the first pivot. This results in fill-in in rows four and six. However, if pivots are chosen in the natural order in which they appear, no fill-in is generated.

Duff, et. al. [10] also mention other pivot selection strategies. However, in practice the Markowitz strategy and minimum degree seem to be about as good as the heuristics get. In addition, we can note that the minimum degree algorithm and the Markowitz strategy do not tell us which entry to choose as the next pivot when multiple entries tie for the lowest count.

3.6. Banded Matrices

Some matrices have special forms that ensure fill-in is globally confined throughout the factorization process. A banded matrix is such a form. Non-zero elements of a banded matrix lie at fixed distances to the right and left of the matrix's diagonal. The ice sheet model without pressure generates banded matrices. Figure 3.21 illustrates the pattern of non-zero entries in a banded matrix.

```
x x x
x x x x
x x x x x
  x x x x x
    x x x x x
      x x x x x
        x x x x x
          x x x x x
            x x x x
              x x x
```

Figure 3. 21. A banded 9x9 matrix.

The maximum number of non-zero entries to the right of a diagonal entry is called the upper bandwidth, m_u, and the maximum number of elements to the left of a diagonal entry is called the lower bandwidth, m_l. The total bandwidth of the matrix is $m_l + m_u + 1$. For a symmetric matrix, $m_l = m_u$. If the diagonal pivots are stable and Gaussian elimination is performed without row interchanges, then all fill-in occurs within the band. If pivoting is performed with Gaussian elimination, fill-ins are restricted to an upper band that is no wider than $m_l + m_u$ and a lower bandwidth that is no wider than m_l, giving a total bandwidth that is no wider than $2m_l + m_u$. It is easy to see that row interchanges never increase the lower bandwidth because row interchanges never introduce non-zero entries to the left of the left-most non-zero entry in any row. A row interchange can increase the upper bandwidth to as much as $m_l + m_u$. Simply consider interchanging the first and third rows in Figure 3.21. If the upper bandwidth of the pivot row is increased, to $m_l + m_u$, row replacement will cause the upper bandwidth of the following m_l rows to increase to no more than $m_l + m_u - 1$ and any subsequent row interchange will continue to have an upper bandwidth bound to $m_l + m_u$.

A banded matrix can be stored in a two-dimensional array without storing the zero entries of the matrix that appear outside the band. For each row of the matrix we store the entries before the diagonal in array columns 1 through m_l, the diagonal entry in array column $m_l + 1$, and the entries to the right of the diagonal entry in array columns $m_l + 2$ through $m_l + m_u + 1$. If row interchanges will be performed, then an additional m_l columns are allocated in the array to accommodate them. Entries of the matrix are directly accessible in the array using the row number and a mapped column number. If i is the row of the entry to be retrieved and j is the column, then i is the row of the array containing the entry and $j - i + m_l + 1$ is the column of the array containing the entry.

If the bandwidth is small compared to the number of columns in the array, the banded storage scheme is very efficient. For the rectangular coordinates of the ice-sheet model and 3 degrees of freedom we have already shown that $m_l = m_u = 3(xy + x + 1) + 2$ where x is the number of nodes in the first dimension, y is the number of nodes in the second dimension. For a rectangular region that is 5x40x40 nodes, the total bandwidth is 1,241. The number of columns in **A** for this problem is 5x40x40x3=24,000. Thus the banded matrix can be stored in a space that is only 5.2% of the size of the full matrix.

If we take the same rectangular region and instead label it with the dimensions 40x40x5, the total bandwidth of the matrix becomes 9,845. We are still solving the same problem, and we should get an identical solution except for some permutations of the solution vector. This indicates that there should be a set of permutations that transforms this statement of the problem to the prior statement and reduces the bandwidth of **A** accordingly. The conversion of the problem from a 5x40x40 rectangular system to a 40x40x5 rectangular system is nothing more than a mapping of the node numbers from one coordinate system to another. A node l is characterized by having its associated coefficients appear in column l of **A** and having its characteristic equation defined by the values that appear in row l of **A** and **b**. Mapping this node number to node number m means interchanging rows l and m in **A** and **b**, and interchanging columns l and m in **A**. This type of permutation is a *symmetric permutation*. If **A** is symmetric before a symmetric permutation is made, it is symmetric after a symmetric permutation is made. For a symmetric matrix all

entries on row l or column l obey the relation $a_{lh} = a_{hl}$. After row and column interchanges a_{lh} is mapped to a_{mh} and a_{hl} is mapped to a_{hm}. By transitivity we have $a_{mh} = a_{hm}$ for all h, so symmetry has been maintained for entries in row l and column l. The same symmetry argument applies for all entries that are initially in row m or column m.

There are two questions that immediately come to mind.

1. How small can the bandwidth be made?
2. How does the number of arithmetic operations required to solve a problem vary for differing permutations with differing bandwidths?

This thesis does not explore these bandwidth-changing permutations further, but it might be interesting to do so. An alternative question is "What node numbering scheme minimizes the bandwidth of **A**?" Another possible approach to the same central issue is "How does one prove or disprove that numbering the nodes of a rectangular region in the order of smallest dimension first, followed by next smallest dimension second, followed by largest dimension last minimizes the bandwidth of **A** over all possible numbering schemes?"

While Gaussian elimination with row interchanges on a banded matrix has tight bounds on the lower and upper bandwidths, the lower bandwidth is not so tightly bound for **LU** factorization. In the worst case, the bound on the lower bandwidth of **L** is $n-1$ where n is the number of columns in **A**. This simple fact was discovered while attempting to write a program to do banded **LU** factorization with row interchanges. The program would fail as it tried to access entries to the left of the lower band. This behavior is easy to understand using Figure 3.21 as an illustration. Suppose the first row is interchanged with the second row in the first stage of factorization. A non-zero entry is generated at row 2, column 1 as a component of **L**. Now suppose at the second stage of factorization, the second row is interchanged with the third row. Now, there is a non-zero entry at row 3, column 1 that is a component of **L**. If row k is interchanged with row $k+1$ at every stage k, then there is a non-zero entry at row n, column 1 that is a component of **L** and the

lower bandwidth of the factored matrix is $n-1$. This could be an issue for problems where we wish to solve $\mathbf{Ax} = \mathbf{b}$ with multiple right hand sides.

In addition to the banded matrix illustrated by Figure 3.21, there can also be variable-band matrices as illustrated in the next figure.

```
x x x
x x x x
x x x x   x
  x x x x x x x
      x x x x
        x x x x
      x x x x x   x
            x x x x
              x x x
                x x
```

Figure 3. 22. Variable-band matrix.

Like banded matrices, the essential feature of variable-band matrices is that row replacements do not introduce non-zero elements outside the band if no permutations are made. Variable-band matrices have also been called skyline, profile, and envelope matrices.

Duff, et. al. discuss other matrix forms that have desirable characteristics when solving $\mathbf{Ax} = \mathbf{b}$ by Gaussian elimination. Such forms are block tridiagonal, doubly bordered block diagonal, and bordered block triangular. While some of these forms have characteristics that are similar to matrices generated by the ice sheet model, including matrices that have pressure terms on the right side and bottom of the matrix, it was beyond the scope of this thesis to carefully investigate them. My intuition is that it is probably more efficient and more productive to investigate software packages for solving general sparse systems of equations than it is to investigate these special forms and to use them to write software for solving ice sheet problems. In addition to presenting special matrix forms, Duff, et. al. also present methods that order problems for small bandwidth. For the rectangular ice-sheet model, none of these methods were simpler or produced smaller bandwidths than simply numbering nodes along the smallest dimension first, along the next smallest dimension second, and along the longest dimension last.

Nonetheless, the special matrix forms have two important attributes. First, by their design, they limit fill-in at a global level. This is opposed to limiting fill-in at the local level at each stage of the factorization as the Markowitz strategy and the minimum degree algorithm do. This gives us the opportunity to know what the memory requirements will be before the problem is solved. Second, they tend to use 2-dimensional arrays for data storage instead of the sparse data structures described earlier. The sparse data structures inherently use indirect addressing which limits the applicability of vector processors and parallel computers. Directly addressable arrays tend to be less constraining when parallelizing algorithms.

3.7. Frontal Methods

This section borrows heavily from the discussion of frontal methods presented by Duff, et. al. I choose to present it for two reasons. One, frontal methods have their origins in solving finite element problems and the ice sheet model relies on the finite element method. Second, one of the software packages used in this work uses multi-frontal methods.

Duff, et. al. point out that frontal methods are most easily understood in the context of finite element problems even though they are applicable to other matrices, too. To begin, consider the triangulated region in the following figure.

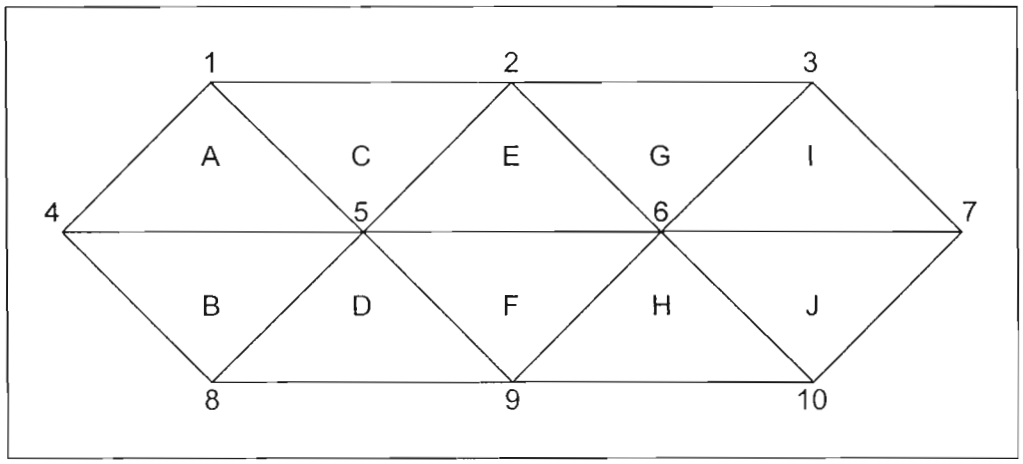

Figure 3. 23. A triangulated FEM region.

The finite element method considers the lettered triangular regions to be elements and the numbered vertices to be nodes. The nodes are associated with one of more variables. For our example, we will assume each node has one degree of freedom and is thus associated with one variable. For each element, FEM generates an element matrix that contains entries for each node and for each possible cross product of nodes. Thus element *A* generates a 3x3 element matrix for variables 1, 4, and 5. Similarly, element *B* generates a 3x3 element matrix for variables 4, 5, and 8. When done, FEM generates a matrix **A** that is the sum of the element matrices. The essence of the frontal method is that **A** is generated as the element matrices are formed and summed, and **LU** factorization is performed on variables as soon as they are totally summed. The result is that **A** is divided into 3 regions: variables that have been fully summed and factored, variables that are partially summed but not factored, and variables that are not summed. The summation and factorization take place in a 2-dimensional array stored in memory. The portions of **L** and **U** corresponding to the totally summed and factored variables are removed from the working array. They can be stored either in memory or on secondary storage. The partially summed variables are stored in the 2-dimmensional array, and as new element matrices are computed, they are added to the 2 dimensional array. The 2-dimmensional array of partially summed variables is the "front" of the matrix.

Let's take a look at this process for the triangulated region in Figure 3.23. First, matrix element *A* is formed and summed producing the following frontal matrix. The column and row numbers designate the node number, or variable numbers.

```
  1 4 5
1 x x x
4 x x x
5 x x x
```

Next, element *B* is formed and summed resulting in the following frontal matrix.

```
  1 4 5 8
1 x x x
4 x x x x
5 x x x x
8   x x x
```

At this point node 4 is total summed, so it can be factored. The first step is to perform a symmetric permutation that places node 4 in the first row and column. The permutation produces the following.

```
  4 1 5 8
4 x x x x
1 x x x
5 x x x x
8 x   x x
```

Factoring variable 4 produces the following. The "#" symbol denotes fill-in.

```
  4 1 5 8
4 u u u u
1 l x x #
5 l x x x
8 l # x x
```

At this point the portions of **L** and **U** corresponding to variable 4 can be removed from the frontal matrix and stored. Storing **L** and **U** as a collection of sparse vectors with unordered entries is appropriate. The frontal matrix then becomes the following.

```
  1 5 8
1 x x #
5 x x x
8 # x x
```

The process can continue until **A** has been fully summed and factored. Elements should be summed in an order that minimizes the size of the frontal matrix by efficiently forming total sums for the variables in the frontal matrix. There must be a system in place for knowing when a variable is fully summed. Also, the process can proceed more efficiently when the maximum size of the frontal matrix is known a priori so that memory for it only needs to be allocated once.

If **A** is neither symmetric and positive definite nor diagonally dominant, then partial pivoting may be necessary to ensure stability. Partial pivoting may delay the removal of a fully summed variable from the frontal matrix. From the example above, if row 1 for variable 4 needs to be interchanged with row 4 for variable 8, then variable 4 cannot be eliminated. Once variable 8 is fully summed, we can complete the

factorization for it and eliminate it. Once partial pivoting is performed, the frontal matrix is no longer symmetric. However, if partial pivoting is needed, then the symmetry of the frontal matrix isn't a concern anyway.

Permutation vectors that identify the variable numbers stored in each element of the frontal matrix must also be maintained. This is nothing more than storing the variable numbers indicated in the row and column margins in the example above. The i'th element of the permutation vector gives the variable number for the i'th row or column of the frontal matrix.

The frontal method can be used for non-FEM methods as well. Rows of **A** are appended to the frontal matrix one at a time. Rows are fully summed when they enter the frontal matrix because rows of **A** are fully summed. Once the column for a variable has been fully summed, meaning all rows that contain a non-zero entry for the variable have been added to the frontal matrix, the variable can be eliminated and the corresponding components of **L** and **U** can be removed from the frontal matrix and stored. Only the frontal matrix needs to be stored in main memory. **A**, **L**, and **U** can be stored in secondary storage if desired. The frontal method is applicable to vector processors because the frontal matrix is stored as a full, 2-dimensional array that is directly addressable.

The frontal method can be viewed as an assembly tree as shown in the next figure.

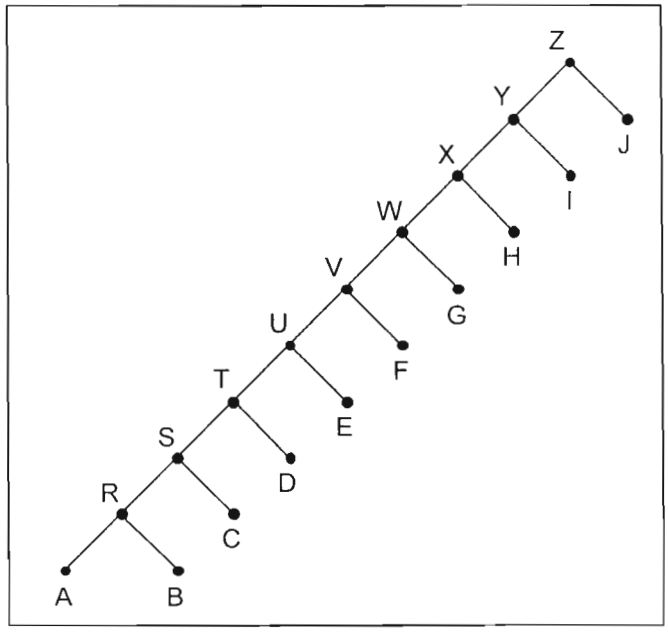

Figure 3. 24. Assembly tree of frontal method.

Elements A and B are assembled to form frontal matrix R. R is reduced and assembled with element C to form frontal matrix S. S is reduced and the procedure continues in a linear fashion until Z is reached. At this point all elements have been assembled, all variables have been summed and eliminated, and the **L** and **U** factors are complete.

It is not necessary for the assembly processes to proceed in a linear fashion with a single frontal matrix. Alternative summation orderings of the element matrices can produce a multi-frontal solution. As an example, consider the rectangular FEM problem in Figure 3.25. As before, the elements are labeled with uppercase letters and the vertices are numbered. Assume that each vertex has a single degree of freedom and corresponds to a single variable. Elements A and B are assembled first and summed. Variables 1 and 2 can be eliminated immediately. The frontal matrix containing variables 3, 6, 7, and 8 is temporarily set aside. Elements E and F are then assembled and summed. Variables 11 and 12 are immediately eliminated leaving a second frontal matrix containing variables 6, 7, 8, and 13. Now the two frontal matrices are summed and variables 6 and 7 are eliminated, leaving another frontal matrix containing variables 3, 8, and 13. Now elements C and D are assembled and summed, variables 4 and 5 are eliminated, and the resulting frontal matrix is set aside. Finally, elements G and H are assembled and summed, and variables 14 and 15

are eliminated. This frontal matrix is then combined with the frontal matrix generated from elements C and D and variables 9 and 10 are eliminated. The two remaining frontal matrices are then combined and the remaining variables 3, 8, and 13 are eliminated.

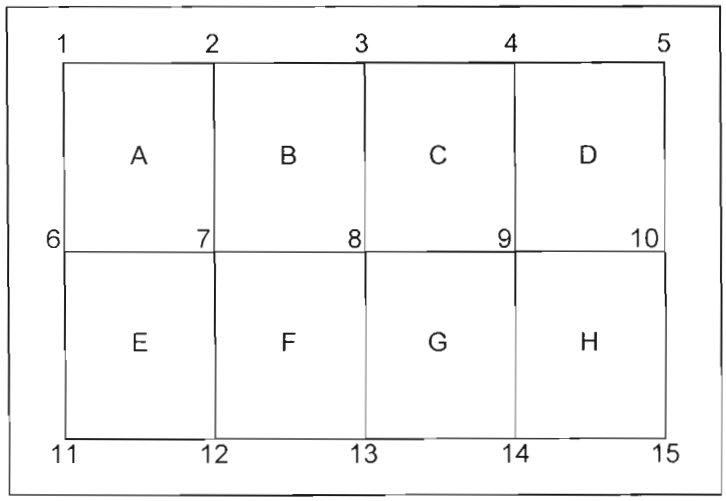

Figure 3. 25. A rectangular FEM region.

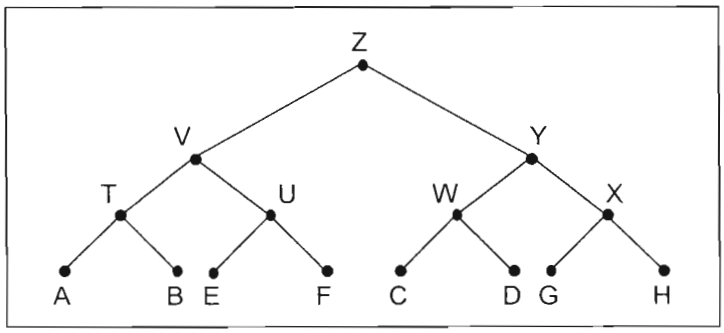

Figure 3. 26. Assembly tree for multi-frontal method.

4. BLAS: Basic Linear Algebra Subprograms

Many basic operations in linear algebra are computationally intensive. While these operations can usually be easily coded in languages such as Fortran or C, computer performance can be improved when they are written in assembly language and coded to make best use of the computer's hardware design. BLAS is a set of basic linear algebra subprograms. The operations performed by BLAS, the subprogram naming conventions, and the calling parameters are defined by a de facto standard. Hardware manufactures generally write BLAS for their computers, but public domain versions for specific machine architectures also exist.

BLAS operations are divided into 3 major categories. BLAS level 1 routines are operations on vectors. The routines include swapping the contents of two vectors, scaling a vector by a constant, adding one vector to another, forming the dot product of two vectors, and computing vector norms. All levels of BLAS have separate routines for different data types. The supported data types are single precision real, double precision real, single precision complex, and double precision complex.

BLAS level 2 routines are matrix-vector products. A typical operation is $\mathbf{y} \leftarrow \mathbf{A}\mathbf{x} + \alpha\,\mathbf{y}$, where $\mathbf{A}$ is a matrix, $\mathbf{x}$ and $\mathbf{y}$ are vectors, and α is a constant. Specific routines are available for various matrix forms such as general, general banded, symmetric, symmetric banded, triangular, triangular banded, hermitian, and hermitian banded. (A hermitian matrix is defined by $a_{ij} = \overline{a_{ji}}$ where $\overline{a_{ji}}$ is the complex conjugate of a_{ji}.) Additional operations include $\mathbf{x} \leftarrow \mathbf{A}^{\mathrm{T}}\mathbf{x}$, outer vector product ($\mathbf{A} \leftarrow \mathbf{x}\mathbf{y}^{\mathrm{T}} + \mathbf{A}$), and triangular solution ($\mathbf{x} \leftarrow \mathbf{A}^{-1}\mathbf{x}$ where $\mathbf{A}$ is upper or lower triangular).

BLAS level 3 routines are matrix-matrix products. A typical operation is $\mathbf{C} \leftarrow \alpha\,\mathbf{A}\mathbf{B} + \mathbf{C}$. As with BLAS level 2, specific routines are available for various matrix forms.

BLAS performance is achieved by carefully controlling the movement of data through the computer. The movement of data within the computer to perform these basic operations can consume a large fraction of the total execution time. Data residing in the computer's hardware registers can be accessed most quickly, so management of register usage is important.

In modern computers, data flowing between main memory and the CPU is buffered in cache memory. The CPU can access data much more quickly if the data can be retrieved from cache. However, the cache is relatively small compared to main memory. When data is accessed that is not in the cache, it is retrieved from main memory and stored in cache. To store the new data in cache means that older data must be dropped from the cache. BLAS routines are carefully written to maximize the number of memory references that can be satisfied by data that is already in cache, thus optimizing efficiency.

Most modern day computers use a memory management technique called virtual memory. With virtual memory a program potentially has the entire address space of the computer at its disposal, even when the amount of physical main memory in the computer is less than the size of the address space. For example, a 64-bit computer has an address space of 2^{64} bytes, but we do not see computers with this amount of main memory. To implement virtual memory the program's address space is divided into blocks of addresses called *pages*. Page sizes of 4 kilobytes to 64 kilobytes are common. In addition, the computer's memory is divided into blocks called *page frames*. The size of a page is the same as the size of a page frame. A page of the program can be placed in any available page frame and pages of the program that are not in use can be stored on disk in the *page file*. The location of each page is tracked in the *page table* that is stored in memory. Each process running on the computer has its own page table. The operating system manages the storage of pages and maintains the page table. When the program references an address, the computer's memory management hardware must convert the virtual address specified by the program to the physical address in memory by using the storage map contained in the page table. In practice, it would be much too slow if the computer had to resolve each and every virtual address by accessing the page table. Instead, the hardware maintains a small associative memory called the *translation look-aside buffer (TLB)* that stores the most recently accessed page table entries. Because the TLB is associative, it can look at all its entries at

once when presented with a virtual address and return the one page table entry corresponding to the virtual address. If the page table entry is in the TLB, the virtual address is mapped to a physical address very quickly. If the page table entry is not in the TLB, the page table must be accessed from memory at much greater expense. A TLB typically has capacity to store 64 page table entries. The BLAS can be designed to maximize TLB hits.

Another issue with virtual memory computers is accessing pages of the program that have been stored on disk in the page file. Moving a page from disk to memory, called a *page fault*, has very high overhead. There is nothing that BLAS can do to minimize page faults. When page faults occur excessively, it is an indication that the computer does not have enough physical memory, or is running too many processes for the amount of physical memory that it has.

Since BLAS performance is obtained by carefully managing access times to variables, different performance levels are associated with each level of BLAS operations. BLAS level 1 operations touch $O(n)$ values in $O(n)$ variables. A value is touched every time it appears in the calculation. In a BLAS level 1 vector dot product, each value in the two vectors is touched once. Multiplying two n x n matrices involves n^3 touches of $2n^2$ variables. Matrix-vector multiplication involves $2n^2$ touches of $n^2 + n$ variables. The more times a single variable is touched, the greater the opportunity there is for BLAS to improve performance. Thus the best performance per arithmetic operation is obtained with BLAS level 3 operations, followed by BLAS level 2 operations, with BLAS level 1 operations taking up the rear.

Although newer additions to BLAS are beginning to offer support for sparse vectors and matrices, maximum performance is achieved for dense vectors and matrices stored as 1-dimensional and 2-dimensional arrays. If the software for solving sparse systems of linear equations can be structured to effectively use BLAS subprograms, they can benefit from the performance of BLAS. The results section of this thesis shows as much as a seven fold performance gain when using a well-tuned BLAS versus a C implementation of the BLAS subprograms.

The next two pages are a reprint of the BLAS Quick Reference Guide [2]. The guide summarizes the BLAS operations, subprogram names, and calling parameters.

Meaning of prefixes

S - REAL	C - COMPLEX
D - DOUBLE PRECISION	Z - COMPLEX* (o this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS

In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS

Matrix types:

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Sym. Band	SP - Sym. Packed
HE - HErmitian	HB - Herm. Band	HP - Herm. Packed
TR - TRiangular	TB - Triang. Band	TP - Triang. Packed

Level 2 and Level 3 BLAS Options

Dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRANx	= 'No transpose', 'Transpose', 'Conjugate transpose' (X, X^T, X^H)
UPLO	= 'Upper triangular', 'Lower triangular'
DIAG	= 'Non-unit triangular', 'Unit triangular'
SIDE	= 'Left', 'Right' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANSx = 'T' and TRANSx = 'C' have the same meaning.
For Hermitian matrices, TRANSx = 'T' is not allowed.
For complex symmetric matrices, TRANSx = 'H' is not allowed.

References

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Math. Soft.* 5 (1979) 308-325.

J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* 14.1 (1988) 1-32.

J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* (1990).

Obtaining the Software via `netlib@ornl.gov`

To receive a copy of the single-precision software, type in a mail message:

```
send sblas from blas
send sblas2 from blas
send sblas3 from blas
```

To receive a copy of the double-precision software, type in a mail message:

```
send dblas from blas
send dblas2 from blas
send dblas3 from blas
```

To receive a copy of the complex single-precision software, type in a mail message:

```
send cblas from blas
send cblas2 from blas
send cblas3 from blas
```

To receive a copy of the complex double-precision software, type in a mail message:

```
send zblas from blas
send zblas2 from blas
send zblas3 from blas
```

Send comments and questions to `lapack@cs.utk.edu` .

Basic Linear Algebra Subprograms

A Quick Reference Guide

University of Tennessee
Oak Ridge National Laboratory
Numerical Algorithms Group Ltd.

May 11, 1997

Figure 4. 1. BLAS Quick Reference Guide.

Figure 4.1. BLAS Quick Reference Guide. (Continued)

Level 1 BLAS

	dim	scalar	vector	vector	scalars	5-element array		precisions
SUBROUTINE xROTG (					A, B, C, S)		Generate plane rotation	S, D
SUBROUTINE xROTMG(					D1, D2, A, B,	PARAM)	Generate modified plane rotation	S, D
SUBROUTINE xROT (	N,		X, INCX,	Y, INCY,	C, S)		Apply plane rotation	S, D
SUBROUTINE xROTM (	N,		X, INCX,	Y, INCY,		PARAM)	Apply modified plane rotation	S, D
SUBROUTINE xSWAP (	N,		X, INCX,	Y, INCY)			$x \leftrightarrow y$	S, D, C, Z
SUBROUTINE xSCAL (	N,	ALPHA,	X, INCX)				$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
SUBROUTINE xCOPY (	N,		X, INCX,	Y, INCY)			$y \leftarrow x$	S, D, C, Z
SUBROUTINE xAXPY (	N,	ALPHA,	X, INCX,	Y, INCY)			$y \leftarrow \alpha x + y$	S, D, C, Z
FUNCTION xDOT (	N,		X, INCX,	Y, INCY)			$dot \leftarrow x^T y$	S, D, DS
FUNCTION xDOTU (	N,		X, INCX,	Y, INCY)			$dot \leftarrow x^T y$	C, Z
FUNCTION xDOTC (	N,		X, INCX,	Y, INCY)			$dot \leftarrow x^H y$	C, Z
FUNCTION xxDOT (	N,		X, INCX,	Y, INCY)			$dot \leftarrow \beta + x^T y$	SDS
FUNCTION xNRM2 (	N,		X, INCX)				$nrm2 \leftarrow \|x\|_2$	S, D, SC, DZ
FUNCTION xASUM (	N,		X, INCX)				$asum \leftarrow \|re(x)\|_1 + \|im(x)\|_1$	S, D, SC, DZ
FUNCTION IxAMAX(	N,		X, INCX)				$amax \leftarrow \|re(x_i)\| + \|im(x_i)\|$ $= max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

	options			dim	b-width	scalar	matrix	vector	scalar	vector		
xGEMV (		TRANS,		M, N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A = m \times n$	S, D, C, Z
xGBMV (		TRANS,		M, N,	KL, KU,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A = m \times n$	S, D, C, Z
xHEMV (	UPLO,			N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHBMV (	UPLO,			N,	K,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xHPMV (	UPLO,			N,		ALPHA,	AP,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	C, Z
xSYMV (	UPLO,			N,		ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSBMV (	UPLO,			N,	K,	ALPHA,	A, LDA,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xSPMV (	UPLO,			N,		ALPHA,	AP,	X, INCX,	BETA,	Y, INCY)	$y \leftarrow \alpha Ax + \beta y$	S, D
xTRMV (	UPLO,	TRANS,	DIAG,	N,			A, LDA,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTBMV (	UPLO,	TRANS,	DIAG,	N,	K,		A, LDA,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTPMV (	UPLO,	TRANS,	DIAG,	N,			AP,	X, INCX)			$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
xTRSV (	UPLO,	TRANS,	DIAG,	N,			A, LDA,	X, INCX)			$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTBSV (	UPLO,	TRANS,	DIAG,	N,	K,		A, LDA,	X, INCX)			$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
xTPSV (	UPLO,	TRANS,	DIAG,	N,			AP,	X, INCX)			$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z

	options	dim	scalar	vector	vector	matrix		
xGER (		M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^T + A, A = m \times n$	S, D
xGERU (		M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^T + A, A = m \times n$	C, Z
xGERC (		M, N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^H + A, A = m \times n$	C, Z
xHER (	UPLO,	N,	ALPHA,	X, INCX,		A, LDA)	$A \leftarrow \alpha xx^H + A$	C, Z
xHPR (	UPLO,	N,	ALPHA,	X, INCX,		AP)	$A \leftarrow \alpha xx^H + A$	C, Z
xHER2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xHPR2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	AP)	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
xSYR (	UPLO,	N,	ALPHA,	X, INCX,		A, LDA)	$A \leftarrow \alpha xx^T + A$	S, D
xSPR (	UPLO,	N,	ALPHA,	X, INCX,		AP)	$A \leftarrow \alpha xx^T + A$	S, D
xSYR2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	A, LDA)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
xSPR2 (	UPLO,	N,	ALPHA,	X, INCX,	Y, INCY,	AP)	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Level 3 BLAS

	options			dim	scalar	matrix	matrix	scalar	matrix		
xGEMM (		TRANSA,	TRANSB,	M, N, K,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C = m \times n$	S, D, C, Z
xSYMM (	SIDE, UPLO,			M, N,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C = m \times n, A = A^T$	S, D, C, Z
xHEMM (	SIDE, UPLO,			M, N,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C = m \times n, A = A^H$	C, Z
xSYRK (	UPLO,	TRANS,		N, K,	ALPHA,	A, LDA,		BETA,	C, LDC)	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C = n \times n$	S, D, C, Z
xHERK (	UPLO,	TRANS,		N, K,	ALPHA,	A, LDA,		BETA,	C, LDC)	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C = n \times n$	C, Z
xSYR2K(	UPLO,	TRANS,		N, K,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C = n \times n$	S, D, C, Z
xHER2K(	UPLO,	TRANS,		N, K,	ALPHA,	A, LDA,	B, LDB,	BETA,	C, LDC)	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C = n \times n$	C, Z
xTRMM (	SIDE, UPLO,	TRANSA,	DIAG,	M, N,	ALPHA,	A, LDA,	B, LDB)			$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B = m \times n$	S, D, C, Z
xTRSM (	SIDE, UPLO,	TRANSA,	DIAG,	M, N,	ALPHA,	A, LDA,	B, LDB)			$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B = m \times n$	S, D, C, Z

5. Software Packages for Solving Sparse Systems of Linear Equations

The next three pages list software packages that are freely available on the Web. One can visit http://www.netlib.org/utk/people/JackDongarra/la-sw.html to view the list and hyperlink to the various packages. The list is compiled by Jack Dongarra, Distinguished University Professor, University of Tennessee, Department of Computer Science. Dongarra has a long history of work in computational linear algebra and his name often appears in this field. For example, he is one of the references listed on the BLAS Quick Reference Guide presented in the previous chapter. One can learn more about Dongarra and find other useful links at http://www.netlib.org/utk/people/JackDongarra/.

FREELY AVAILABLE SOFTWARE FOR LINEAR ALGEBRA ON THE WEB (May 2004)

Here is a list of freely available software for the solution of linear algebra problems. The interest is in software for high-performance computers that's available in "open source" form on the web for solving problems in numerical linear algebra, specifically dense, sparse direct and iterative systems and sparse iterative eigenvalue problems. Please let me know about updates and corrections.

Additional pointers to software can be found at:
http://www.nhse.org/rib/repositories/nhse/catalog/#Numerical_Programs_and_Routines
A survey of Iterative Linear System Solver Packages can be found at:
http://www.netlib.org/utk/papers/iterative-survey
Thanks, Jack

Software Package	Support	Type		Language			Mode		Dense	Sparse Direct		Sparse Iterative		Sparse Eigenvalue	
SUPPORT ROUTINES		Real	Complex	f77	c	c++	Seq	Dist		SPD	Gen	SPD	Gen	Sym	Gen
ATLAS	yes	X	X	X	X		X		X						
BLAS	yes	X	X	X	X		X		X						
FLAME	yes	X	X	X	X		X		X						
LINALG++	?														
MTL	yes	X				X	X		X						
NEWMAT	yes	X				X	X		X						
NISTSPARSEBLAS	yes	X	X	X	X		X								
PSBLAS	yes	X	X	X	X		X	M							
Sparsel++	yes	X	X		X	X	X								
uBLAS	yes	X	X		X	X	X		X						
DIRECT SOLVERS		Real	Complex	f77	c	c++	Seq	Dist		SPD	Gen	SPD	Gen	Sym	Gen
LAPACK	yes	X	X	X	X		X		X						
LAPACK95	yes	X	X	95			X		X						
NAPACK	yes	X		X			X		X			X		X	
PLAPACK	?	X	X	X	X			M	X						
PRISM	no	X		X			X	M	X						
ScaLAPACK	yes	X	X	X	X			M/P	X						
SPARSE DIRECT SOLVERS		Real	Complex	f77	c	c++	Seq	Dist		SPD	Gen	SPD	Gen	Sym	Gen
DSCPACK	yes	X			X		X	M		X					
HSL	yes	X	X	X			X			X	X				
MFACT	yes	X			X		X	M		X					
MFAMPS	yes	X	X	X	X		X	M		X	X				
PSPASES	yes	X		X	X			M		X					
SPARSE	?	X	X		X		X			X	X				
SPOOLES	?	X	X		X		X	M			X		X		
SuperLU	yes	X	X	X	X		X	M			X				
TAUCS	yes	X	X		X		X			X	X	X	X		
UMFPACK	yes	X	X		X		X				X				
Y12M	?	X		X			X			X	X				

http://www.netlib.org/utk/people/JackDongarra/la-sw.html 10/9/2005

Figure 5. 1. Freely available software for linear algebra on the Web.

PRECONDITIONERS		Real	Complex	f77	c	c++	Seq	Dist		SPD	Gen	SPD	Gen	Sym	Gen
BPKIT	yes	X		X	X	X	X	M							
PARPRE	yes	X			X			M							
SPAI	yes	X			X		X	M							
SPARSE ITERATIVE SOLVERS		Real	Complex	f77	c	c++	Seq	Dist		SPD	Gen	SPD	Gen	Sym	Gen
BILUM	no	X		X			X					X	X		
BlockSolve95	?	X		X	X	X		M				X	X		
CLRLACS	yes	X	X	X			X					X	X		
GMM++	yes	X	X			X	X			X	X	X	X		
HYPRE	yes	X		X	X		X	M				I	P	P	
IML++	?	X		X	X	X	X					X	X		
LLL	yes	X				X	X					X	X		
ITPACK	?	X		X			X					X	X		
LASPack	yes	X			X		X					X	X		
LSQR	yes	X		X	X		X						X		
pARMS	yes	X		X	X		X	M				X	X		
PLTSs	yes	X	X	X	X		X	M				X	X		
PIM	yes	X	X	X			X	M				X	X		
P-SparsLIB	yes	X		X				M					X		
QMRPACK	?	X	X	X			X					X	X	X	X
SLAP	?	X		X								X	X		
SPLIB	?	X		X			X					X	X		
SYMMLQ	yes	X		X			X					X	X		
Trilinos	yes	X	X		X		X	X		X	X	P	P		
Templates	yes	X		X	X		X					X	X		
SPARSE EIGENVALUE SOLVERS		Real	Complex	f77	c	c++	Seq	Dist		SPD	Gen	SPD	Gen	Sym	Gen
LZPACK	yes	X	X	X			X	M/P						X	
LASO	?	X		X			X							X	
PARPACK	yes	X	X	X	X	X	X	M/P						X	X
PLANSO	yes	X		X			X	M						X	
SLEPc	yes	X	X		X		X	M						X	X
SPAM	yes	X		90			X								X
TRLAN	yes	X		X			X	M						X	

LINALG *: This is a collection of software that is available but too varied to describe.

Notes:
- Type:
  - Real = Real arithmetic
  - Complex = Complex arithmetic
- Support: An email address where you can send questions and bug reports.
- Language: f77(Fortran 95), c, c++
- Mode:

http://www.netlib.org/utk/people/JackDongarra/la-sw.html 10/9/2005

Figure 5.1. Freely available software for linear algebra on the Web. (Continued)

Seq = Sequential, vector and/or SMP/multithreaded versions
Dist = distributed memory message passing (M = MPI, P = PVM)
Dense: Dense, triangular, banded, tridiagonal matrices
Sparse: A sparse matrix representation is used to contain the data.
Direct: A direct approach is used to factor and solve the system.
SPD: The matrix is symmetric and positive definite
Gen: The matrix is general
Iterative: An iterative method is used to solve the system.
SPD: The matrix is symmetric and positive definite
Gen: The matrix is general
P indicates preconditioners
Sparse eigenvalue: An iterative method is used to find some of the eigenvalues
Sym: The matrix is symmetric (Hermitian in the complex case)
Gen: The matrix is general

http://www.netlib.org/utk/people/JackDongarra/la-sw.html 10/9/2005

Figure 5.1. Freely available software for linear algebra on the Web. (Continued)

Two classes of solvers for sparse linear systems are listed: direct solvers and iterative solvers. The focus of this work is on sparse direct solvers, which are based on Gaussian elimination and **LU** factorization.

Iterative solvers begin with an approximate solution and through a series of iterative steps refine the approximation until suitable accuracy has been obtained. A straightforward iterative method is the Jacobi method where the k'th iteration produces an improved estimate from the previous iteration by calculating

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b - \sum_{j \neq i} a_{ij} x_j^{(k-1)}\right). \tag{5.1}$$

The Gauss-Seidel method is similar, but uses estimates from the current iteration as well as the previous iteration as shown in equation 5.2.

$$x_i^{(k)} = \frac{1}{a_{ii}}\left(b - \sum_{j<i} a_{ij} x_j^{(k)} - \sum_{j>i} a_{ij} x_j^{(k-1)}\right). \tag{5.2}$$

Convergence is an important issue for the iterative methods: will a method converge for a given system and how quickly will it converge? It can be shown that the Jacobi method converges if **A** is diagonally dominant. The Gauss-Seidel method is less restrictive. It can be shown that Gauss-Seidel converges if **A** is symmetric and positive definite. The 3-D ice sheet model does not produce diagonally dominant matrices, but it does produce symmetric matrices that may be positive definite. There are also other iterative methods mentioned by Golub and Van Loan [11]. Determining if any of the iterative methods are applicable to the ice-sheet model and implementing applicable software packages could be an additional project for someone.

At an early stage, two software packages were chosen for evaluation: SuperLU and UMFPACK. The packages were found through Web searches for sparse matrix solvers. In hindsight, these proved to be good choices. Making the choices early in this project was also a good decision. At the outset of this work there was no idea how much would be discovered about this subject. Choosing the software packages early and working with them from the beginning provided a firm foundation for the discovery process.

In addition to the sparse direct solvers, a high performance BLAS produced by Kazushige Goto, Visiting Scientist, University of Texas, was used. Goto's BLAS is copyrighted by The University of Texas, 2005, all rights reserved. It is available free of charge for academic purposes. The BLAS is available for several hardware architectures and is recommended in the UMFPACK package. One can learn more about Goto's BLAS from the Texas Advanced Computing Center's website at http://www.tacc.utexas.edu/resources/software.

5.1. SuperLU

SuperLU is a library of ANSI C subroutines for solving sparse linear systems of equations [8][9][13]. The principal developers are Xiaoye (Sherry) Li, Computer Scientist, Lawrence Berkeley National Laboratory; James Demmel, Professor of Computer Science and Mathematics, University of California at Berkeley; and John Gilbert, Professor of Computer Science, University of California at Santa Barbara. The SuperLU libraries are freely available for commercial and non-commercial use. The whole SuperLU software is copyrighted by The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from the U.S. Department of Energy), all rights reserved.

SuperLU is applicable to unsymmetric as well as symmetric matrices. It uses **LU** factorization with threshold pivoting. SuperLU comes in three versions. Sequential SuperLU, known simply as SuperLU, is for single processor computers. Multithreaded SuperLU, known as SuperLU-MT is designed for shared memory multiprocessors. Distributed SuperLU, known as SuperLU-DIST, is for distributed memory parallel computers. Sequential SuperLU version 3.0 was used in this work. It is the latest version of sequential SuperLU at this time.

A shared memory multiprocessor is a parallel computer that allows all processors to access any main memory location. Access to main memory by the processors is coordinated by the computer's hardware. The authors claim that SuperLU-MT can effectively support 16 to 32 processors for sufficiently large matrices. It uses POSIX threads to coordinate processes.

Each processor in a distributed memory computer has its own memory. A communications network between the processors is used to share data and coordinate activities. SuperLU-DIST uses Message Passing Interface (MPI) for interprocess communications. MPI is a common standard for distributed memory parallel computers. The authors claim these versions are designed to make optimum use of the sparsity of **A** and the computer's architecture with attention given to optimum use of cache memory and parallelism.

The overall scheme of SuperLU is to do an **LU** factorization of $\mathbf{P}_r\mathbf{D}_r\mathbf{A}\mathbf{D}_c\mathbf{P}_c$ and use forward substitution and backward substitution to solve for $\mathbf{x}$ in $\mathbf{Ax} = \mathbf{b}$. Matrix $\mathbf{P}_r$ is a row permutation matrix for maintaining stability. Matrix $\mathbf{P}_c$ is a column permutation matrix for maintaining sparsity. Matrices $\mathbf{D}_r$ and $\mathbf{D}_c$ are row and column scaling matrices for conditioning **A** so as to minimize the sensitivity of $\mathbf{A}^{-1}$ to perturbations. SuperLU computes each of these four matrices with various levels of control available to the user. Because SuperLU uses LU factorization, it can compute **x** for multiple right hand sides.

To solve a system of equations SuperLU uses

$$\mathbf{A} = \mathbf{D}_r^{-1}\mathbf{P}_r^{-1}\mathbf{LUP}_c^{-1}\mathbf{D}_c^{-1} \tag{5.3}$$

giving

$$\begin{aligned}\mathbf{x} &= \left(\mathbf{D}_r^{-1}\mathbf{P}_r^{-1}\mathbf{LUP}_c^{-1}\mathbf{D}_c^{-1}\right)^{-1}\mathbf{b} \\ &= \mathbf{D}_c\mathbf{P}_c\mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}_r\mathbf{D}_r\mathbf{b}\end{aligned} \tag{5.4}$$

This equation is solved for **x** by

1. Scaling rows of **b** by $\mathbf{D}_r$
2. Permuting rows of **b** by $\mathbf{P}_r$
3. Using forward substitution with **L** and the scaled and permuted **b** to compute an intermediate vector **y**
4. Using backward substitution with **U** and **y** to compute **x**
5. Permuting rows of **x** by $\mathbf{P}_c$
6. Scaling rows of **x** by $\mathbf{D}_c$

In SuperLU terminology, driver routines are the user callable routines for performing major tasks. SuperLU and SuperLU-MT have two driver routines for solving systems of linear equations: the simple driver and the expert driver. The simple driver chooses $\mathbf{P}_c$ to minimize fill-in. It then computes **L** and **U**.

$\mathbf{P}_r$ is computed as a by-product of threshold pivoting. The driver then solves for $\mathbf{x}$ using $\mathbf{P}_c$, $\mathbf{P}_r$, $\mathbf{L}$, and $\mathbf{U}$.

The expert driver, which is also available in SuperLU-DIST, performs the following steps.

1. Equilibrate $\mathbf{A}$ by computing the row and column scaling matrices $\mathbf{D}_r$ and $\mathbf{D}_c$ so that $\overline{\mathbf{A}} = \mathbf{D}_r \mathbf{A} \mathbf{D}_c$ is better conditioned than $\mathbf{A}$.
2. Preorder rows of $\overline{\mathbf{A}}$ for stability. This is only done in SuperLU-DIST and is called static pivoting. The interprocess communication required to perform threshold pivoting is not practical on a distributed memory parallel computer.
3. Order the columns of $\overline{\mathbf{A}}$ to optimize the sparsity of $\mathbf{L}$ and $\mathbf{U}$ and increase parallelism for SuperLU-MT and SuperLU-DIST.
4. Compute the $\mathbf{LU}$ factorization. Threshold pivoting is done in SuperLU and SuperLU-MT.
5. Solve the system of equations.
6. Perform iterative refinement to improve the solution.
7. Compute error bounds.

Threshold pivoting is implemented as follows. If SuperLU is choosing the i'th pivot, it first determines the value in the i'th column, rows i through n that has the largest absolute value. Let this value be $\left| a_{mi}^{(i)} \right|$. If $\left| a_{ii}^{(i)} \right| \geq u \left| a_{mi}^{(i)} \right|$ where u is a user chosen threshold between 0 and 1, then $a_{ii}^{(i)}$ is used as the pivot. Otherwise, $a_{mi}^{(i)}$ is used as the pivot. The tradeoff is maintaining stability of the calculation versus minimization of fill-in. Threshold pivoting is equivalent to partial pivoting when u=1. If u=0, pivoting is only performed when $a_{ii}^{(i)}$ is zero. A common value for u is 0.1. Static pivoting, on the other hand, determines row permutations from the values a_{ij} before any factorization is performed.

SuperLU can compute the componentwise relative backward error *BERR* discussed in Chapter 2. The meaning of *BERR* is that $\bar{\mathbf{x}}$, the computed value of **x**, is the exact solution of the perturbed linear system of equations $(\mathbf{A}+\mathbf{E})\bar{\mathbf{x}} = \mathbf{b}+\mathbf{f}$ where

$$|e_{ij}| \le BERR \cdot |a_{ij}| \tag{5.5}$$

and

$$|f_i| \le BERR \cdot b_i \tag{5.6}$$

for all i and j. In addition, SuperLU can estimate a forward error bound *FERR* such that

$$\|\mathbf{x}-\bar{\mathbf{x}}\|_\infty / \|\mathbf{x}\|_\infty \le FERR\,. \tag{5.7}$$

When the problem is poorly scaled *FERR* tends to give the relative error of the largest component of **x**, while smaller components of **x** may have significantly higher relative errors. The distributed version of SuperLU does not compute *FERR*. The authors claim that by combining static pivoting with row and column scaling and iterative refinement, the distributed algorithm is as stable as partial pivoting for most matrices observed in actual applications. In cases where computations are not stable, *BERR* provides an indication of a problem.

The driver routines make calls to lower level computational routines to perform tasks such as equilibrating **A**, determining column order, factoring **A**, and performing forward and backward substitution. For large matrices, factorization generally takes most of the time, but choosing the column ordering can also be time consuming.

Matrix **A** must be presented to the driver routine as a C structure defined by SuperLU as `SuperMatrix`. A SuperLU routine takes **A** in either compressed column format or compressed row format and creates the `SuperMatrix` structure. The right hand side of the system of equations, **b**, may be presented to the driver routine as a dense vector if there is only a single right hand side, or as a dense matrix in column major order if there are multiple right hand sides. The solution **x** overwrites **b**.

SuperLU needs a high-performance BLAS to obtain maximum performance. It organizes **A** into supernodes. A supernode is a range of columns of **L** such that the triangular block of **L** below the diagonal is completely filled. In addition, each row of **L** within this range of columns either has all zero entries or all non-zero entries. Because the supernodes are not necessarily symmetric, the **U** portion of the supernode does not have the same dense pattern as **L**. The majority of SuperLU's computation is updating the unfactored submatrix of the supernode using the following block mode update.

$$\mathbf{A}(I,J) \leftarrow \mathbf{A}(I,J) - \mathbf{L}(I,K)\mathbf{U}(K,J) \tag{5.8}$$

A is the unfactored portion of the supernode. **L** and **U** are the factored portions of the supernode. I is the range of rows of the unfactored portion and J is the range of columns of the unfactored portion. K is the number of columns of **L** in the supernode and the number of rows of **U** in the supernode. This looks like a BLAS level 3 operation, and that is in fact what SuperLU-DIST uses. SuperLU and SuperLU-MT work a little differently. The authors claim that the non-zero portions of **U** are dense vectors of varying length. Instead of using BLAS level 3, care is taken to ensure that **L** is loaded into cache once and then BLAS level 2 matrix-vector multiplies are performed for each vector of **U**. The authors refer to this as BLAS level 2.5.

There are five choices for column ordering heuristics for both the simple and expert drivers. They are:

1. Natural order. No column permutations are performed.
2. Multiple Minimum Degree (MMD) applied to the symmetric structure $\mathbf{A}^T\mathbf{A}$
3. Multiple Minimum Degree (MMD) applied to the symmetric structure $\mathbf{A}^T + \mathbf{A}$
4. Column Approximate Minimum Degree (COLAMD)
5. User supplied $\mathbf{P}_c$

Multiple minimum degree is a modified version of the minimum degree algorithm by Joseph Lui [14]. Column approximate minimum degree is another minimum degree like algorithm by Timothy Davis, et. al. [7]. Timothy Davis is the author of UMFPACK, the second software package evaluated in this work. The authors claim that COLAMD is designed for unsymmetric matrices with partial pivoting. It produces

orderings similar to MMD on $\mathbf{A}^T\mathbf{A}$ without explicitly forming $\mathbf{A}^T\mathbf{A}$ and is faster. A user supplied column permutation matrix allows the user to use other column ordering heuristics if desired.

SuperLU has a number of options for speeding up the solution of related systems of equations by reusing information from the prior run. For example, the matrix generated by the ice sheet model has the same pattern of non-zero entries for every time step. This means that column orderings can be computed once for all time steps. Here are the possible options that SuperLU supports.

1. No previous information is used. Factorization is performed from scratch.
2. Reuse $\mathbf{P}_c$. This can be done when the sparsity structure of the matrix remains constant.
3. Reuse $\mathbf{P}_c$, $\mathbf{P}_r$, and the data structures for **L** and **U**. This can be done when the sparsity structure does not change and the entries of **A** are similar from one system to the next so that the row ordering does not need to change.
4. Reuse $\mathbf{P}_c$, $\mathbf{P}_r$, **L**, and **U**. This can be done when the right hand side changes, but **A** remains the same. It can also be used if the changes in **A** are small and iterative refinement converges. This would be an interesting option to try in the ice sheet model. It might significantly reduce execution time.

The **L** and **U** factors generally have many more non-zero entries than **A** due to fill-in. If $\mathbf{P}_r$ and $\mathbf{P}_c$ are not known before factorization begins, then there is no sure way to know how much memory the factors will need. Sequential SuperLU provides three options for memory management.

1. The user can pre-allocate the work area and pass the address and size of the area to the driver routine. If the work area is too small, the driver routine will abort.
2. The user can specify an estimated work size area and the driver routine will initially allocate that amount of memory. If the allocation is insufficient, SuperLU will allocate a new work area, copy

the data into it, and free the original work area. If it cannot allocate a larger work area, then it aborts.

3. SuperLU can estimate the original amount of work area needed. Like option 2, if the estimate is too small, it will allocate a new work area, copy the data into it, and free the original work area. If it cannot allocate a larger work area, then it aborts.

SuperLU-MT memory management is similar. The only difference is that SuperLU-MT will not try to allocate a larger work area if the original work area is too small. This is reasonable in light of the synchronization that would be required between processors if the work area was reallocated.

SuperLU-DIST memory management is completely different. Because $\mathbf{P}_r$ and $\mathbf{P}_c$ are computed before factorization begins, SuperLU-DIST can determine what the fill-in requirements will be a priori and allocate the correct amount of memory.

SuperLU has a number of user options.

1. Factorization. (1) Factor **A** from scratch. (2) Reuse last $\mathbf{P}_c$. (3) Reuse last $\mathbf{P}_r$ and $\mathbf{P}_c$. (4) Reuse last $\mathbf{P}_r$, $\mathbf{P}_c$, **L**, and **U**.
2. Equilibrate **A**. (1) No. (2) Scale rows and columns of **A** to have unit norms.
3. Column ordering. (1) Natural ordering. (2) MMD ordering on $\mathbf{A}^T\mathbf{A}$. (3) MMD ordering on $\mathbf{A}^T + \mathbf{A}$. (4) COLAMD ordering. (5) User specified $\mathbf{P}_c$.
4. Transpose **A**. (1) No. Solve $\mathbf{Ax} = \mathbf{b}$. (2) Yes. Solve $\mathbf{A}^T\mathbf{x} = \mathbf{b}$. (3) Yes. Solve $\mathbf{A}^H\mathbf{x} = \mathbf{b}$ where $\mathbf{A}^H$ is the transpose of **A** with each entry being the complex conjugate of the corresponding entry of **A**.
5. Iterative refinement. (1) No. (2) Single precision iterative refinement. (3) Double precision iterative refinement. (4) Extended precision iterative refinement.
6. Print statistics. (1) No. (2) Yes.

7. Symmetric mode. (1) No. Assume **A** is not diagonally dominant. (2) Yes. Assume **A** is diagonally dominant or nearly so.
8. Pivot threshold. The value of u.
9. Compute reciprocal of pivot growth. (1) No. (2) Yes.
10. Compute condition number of **A**. (1) No. (2) Yes.

SuperLU-DIST has some additional user options. There are also options for tuning the performance of SuperLU. These options relate to blocking sizes and supernode sizes to optimize cache use.

A number of test matrices are available for testing the performance of linear equation solvers. Davis maintains a library of them at the University of Florida. The authors of SuperLU have run benchmarks for a number of matrices [8]. Generally, they find that sequential SuperLU can achieve up to 40% of the theoretical floating-point operations rate on a number of processors. SuperLU-MT demonstrated speedup by factors of 5 to 10 over sequential SuperLU. SuperLU-DIST achieved up to 100 times speedup with a 512-processor Cray T3E.

The authors also did extensive tests comparing SuperLU and UMFPACK. They report that neither package consistently dominated the other in storage cost or time. SuperLU used less memory 60% of the time in a field of 45 matrices. SuperLU took less time for 44% of the matrices when considering both column ordering time and factorization time. When column ordering time is not considered, SuperLU took less time for 77% of the matrices.

The next page is a sample of the output produced by my implementation of the SuperLU expert driver when solving a sample problem from the ice sheet model. The demonstration program begins by reading **A** and **b** from a disk file and storing the data in compressed column format. Expert solver initialization is performed and the selected SuperLU user options are printed. SuperLU has the option to accumulate a number of statistics about the solution process. These statistics are reported next.

```
SuperLU Demonstration
 Rodney Jacobs, University of Maine, 2005

 Initialize compressed column data structures
 # Non-zero Elements........:  1629108
 # Columns..................:  24000
 Read and store matrix A
 Read and store righthand side
 Solve Ax=B
SLUXSOLVE: SuperLU Expert Solver initialization
Fact=DOFACT.........: Factor matrix A from scratch
Equil=YES...........: Scale A's rows and columns to have unit norm
ColPerm=MMD_AT_PLUSA: Use minimum degree column ordering on A'+A
Trans=NOTRANS.......: Solve A * X = B (A is not transposed)
IterRefine=DOUBLE...: Perform double precision iterative refinement
PrintStat=YES.......: Print solver's statistics
SymmetricMode=NO....: Assume A is not diagonally dominant
Diag Pivot Threshold: 1.000000e-01
PivotGrowth=YES.....: Compute reciprocal of pivot growth
ConditionNumber=YES.: Compute reciprocal of condition number

SLUXSOLVE: Solve Ax=b
# columns (rows)............: 24000
# non-zero elements.........: 1629108
Pivot growth...............: 1.175401e+00
Condition number...........: 1.188610e+06
Iterative Refinement Steps..: 2
BERR.......................: 3.833574e-16
FERR.......................: 6.768792e-10
# nonzeros in L............: 7281453
# nonzeros in U............: 7281453
# nonzeros in L+U...........: 14562906
L\U memory (MB)............: 138.328
Total memory needed (MB)....: 142.814
# memory expansions.........: 0

Timings     Time        FLOPs        MFLOPs/sec
-------   -------   ------------   -----------
 Factor     8.11   7.655639e+09        943.98
  Solve     0.10   2.907781e+07        290.78
  Etree     0.03   0.000000e+00          0.00
  Equil     0.07   0.000000e+00          0.00
  Rcond     0.58   0.000000e+00          0.00
 Refine     0.95   0.000000e+00          0.00
         -------   ------------   -----------
  Total     9.84   7.684716e+09        780.97

Wall clock time (seconds)...:  10.028008
 Total CPU time (seconds)...:  10.0199995

 Error Measures
 ---------------------------------------------
 BERR........................:   4.43100082E-16
 ||R||infinity...............:   0.00272948481
 ||H||infinity (lower bound):   4.19542809E-31
 ||A||infinity...............:   1.E+30
 ||B||infinity...............:   1.08198E+11
 ||X||infinity...............:   0.0065058553

 Deallocate memory
SLUSOLVE: free dynamic memory
```

Figure 5. 2. Sample SuperLU output.

The test matrix has 24,000 rows and columns and 1.63×10^6 non-zero entries for a fill-in ratio of 0.283%. Pivot growth remained low throughout the factorization. It is not known how pivot growth is being calculated, so a good interpretation of its value is lacking. The condition number of the matrix is high due to the penalty method being used within FEM. Two steps of iterative refinement were required to bring *BERR* to within the limits of machine round off error. One or two steps of iterative refinement are typical. *FERR* is also low at 6.77×10^{-10}. Because **L** and **U** have the same number of non-zero elements, it appears that factorization took place without row interchanges. **L** and **U** have nearly 9 times the number of non-zero entries as **A**. The total memory needed of 143MB has been well controlled and the fact that zero memory expansions were required means that no inefficiencies were introduced by having a too small workspace. SuperLU's estimate of needed workspace was used in the allocation. The timings are broken down by various activities within SuperLU's process. Most of the time is spent factoring the matrix with iterative refinement coming in as a distant second.

The error measures at the end of the listing are for the original matrix **A** and the computed solution. This set of error measures is common throughout all the test programs in this work. *BERR* has a different value than SuperLU because SuperLU based its value on the scaled matrix. The lower bound on the infinity norm of **H**, the perturbation in **A** for which the computed solution is an exact solution, is quite worthless. One reference had suggested this calculation for approximating the norm. What is really desired is an upper bound on the infinity norm of **H**. An upper bound on the infinity norm of **H** can be computed by multiplying *BERR* by the infinity norm of **A**, giving $4.43 * 10^{14}$. Except for a small number of occurrences of 10^{30} in **A**, most non-zero entries of **A** tend to have values on the order of 10^9, so the upper bound of the infinity norm of **H** doesn't really tell us much about the solution either. Fortunately, *BERR* is a componentwise bound on **H**, so it really appears that we have a good solution.

5.2. UMFPACK

UMFPACK (pronounced umph-pack with two syllables) is a set of ANSI C routines for solving sparse, unsymmetric systems of linear equations. It is written by Timothy Davis of the Computer and Information Sciences Department at the University of Florida. UMFPACK is copyrighted by Davis with all rights reserved. It is freely available at http://www.cise.ufl.edu/research/sparse/umfpack. Personal communication with Davis in the early stages of this work was used to confirm UMFPACK's suitability for solving equations produced by the ice sheet model.

Davis' experience positions him at the crossroads of two major groups of numerical linear algebra researchers. Over the past couple of decades, a lot of work in this field has been done in the United Kingdom under the leadership of Iain Duff starting at the Harwell Laboratory in Oxfordshire and currently at the Rutherford Appleton Laboratory in Oxfordshire. This group is responsible for the Harwell Subroutine Library. In the United States, the Scientific Computing Group at the Lawrence Berkley National Laboratory has developed the SuperLU package. Davis did post-doc work at the European Center for Research and Advanced Training in Scientific Computation under the direction of Iain Duff. He also spent a year on sabbatical as a visiting professor at Stanford in the Scientific Computing / Computational Mathematics Program and a visiting staff member at the Lawrence Berkeley National Laboratory.

UMFPACK uses a multifrontal method and BLAS level 3 routines to perform **LU** factorization. UMFPACK routines are callable from C, FORTRAN, and MATLAB. MATLAB is a high level language with an interactive environment and functions for developing algorithms, data analysis and visualization, and numerical computations. UMFPACK is designed for single processor computers. There are no parallel computer versions. The UMFPACK software works on a variety of UNIX versions including Sun Solaris, Red Hat Linux, IBM AIX, SGI IRIX and Compaq Alpha as well as Microsoft Windows. Version 4.3.1 was the latest version of UMFPACK available at the start of this work and is the version used here.

UMFPACK does an **LU** factorization of $\mathbf{P}_r\mathbf{D}_r\mathbf{A}\mathbf{P}_c$. $\mathbf{P}_r$ is a row permutation matrix for maintaining stability while reducing fill-in. $\mathbf{D}_r$ is a matrix for scaling rows of $\mathbf{A}$ in order to improve the condition of $\mathbf{A}$.

$\mathbf{P}_c$ is a column permutation matrix for reducing fill-in. This approach is analogous to SuperLU with the exception that UMFPACK does not do column scaling. UMFPACK's row scaling is particularly simple. There are three options. (1) No scaling is performed. (2) Each row of **A** is divided by the sum of the absolute values of the entries in that row. (3) Each row of **A** is dividend by the absolute value of the entry in that row with the largest absolute value. Option (2) sets the 1-norm of each row to 1, thus giving $\|\mathbf{A}\|_\infty = 1$. Davis doesn't explain why option 3 might be used. He does clam that scaling is important when using his symmetric strategy and that scaling improves the performance of his unsymmetric strategy. These strategies are discussed a later. Solving $\mathbf{Ax} = \mathbf{b}$ follows the same line of processing as SuperLU.

1. **b** is scaled by $\mathbf{D}_r$.
2. **b** is permuted by $\mathbf{P}_r$.
3. Forward substitution is used with **L** and the scaled and permuted value of **b** to compute an intermediate vector **y**.
4. Backward substitution is used with **U** and **y** to compute **x**.
5. **x** is permuted by $\mathbf{P}_c$.

UMFPACK begins by finding a column ordering for **A** that reduces fill-in without regard for the numerical values of the non-zero entries of **A**. The matrix is scaled and analyzed to determine which of three possible strategies to use for pre-ordering its rows and columns. The available strategies are unsymmetric, 2-by-2, and symmetric. All pivots with zero Markowitz cost are eliminated and placed in the **LU** factors. The following rules are then applied to the remaining submatrix **S** to determine the strategy to use.

1. If **A** is rectangular, then the unsymmetric strategy is used.
2. If the removal of pivots with zero Markowitz cost did not preserve the diagonal of **S**, then the unsymmetric strategy is used.
3. The symmetry σ_1 of **S** is defined as the number of matched off-diagonal entries in **S** divided by the number of off-diagonal entries in **S**. Entries of **S** are the sparse entries defined by the input.

An entry s_{ij} is matched if there is also an entry s_{ji}, even if the values of the two entries are not equal. If $\sigma_1 < 0.1$ then the matrix is very unsymmetric and the unsymmetric strategy is used.

4. If the $\sigma_1 \geq 0.7$ and there are no zeros on the diagonal, then the symmetric strategy is used. **S** is nearly symmetric.
5. The 2-by-2 strategy is attempted. A row permutation $\mathbf{P}_2$ is found that reduces the number of small diagonal entries in **S**. A diagonal entry s_{ii} is considered small if $|s_{ii}| < 0.01 \cdot \max_k |s_{ki}|$. If s_{ii} is small, an attempt is made to find two rows *i* and *j* such that s_{ij} and s_{ji} are large. Swapping these two rows ensures that **S** has large diagonal entries in rows *i* and *j*. Let σ_2 be the symmetry of $\mathbf{P}_2\mathbf{S}$, let d_2 be the number of nonzero diagonal entries of $\mathbf{P}_2\mathbf{S}$, and let the dimension of **S** be ν by ν. If $\sigma_2 > 1.1\sigma_1$ and $d_2 > 0.9\nu$, then use the 2-by-2 strategy. Permuting **S** by $\mathbf{P}_2$ has made the matrix significantly more symmetric.
6. If $\sigma_2 < 0.7\sigma_1$ then use the unsymmetric strategy. The 2-by-2 strategy has significantly worsened the symmetry.
7. If $\sigma_2 < 0.25$ then use the unsymmetric strategy. The matrix is still very unsymmetric.
8. If $\sigma_2 \geq 0.51$ then use the 2-by-2 strategy. The matrix is roughly symmetric.
9. If $\sigma_2 \geq 0.999\sigma_1$ then use the 2-by-2 strategy. The 2-by-2 strategy has improved the symmetry or only made it slightly worse.
10. Otherwise, use the unsymmetric strategy.

The unsymmetric strategy pre-orders the columns of **S** using a modified version of the COLAMD algorithm. COLAMD is a column approximate minimum degree algorithm developed by Davis, Gilbert, Larimore, and Ng [7]. Gilbert is one of the primary developers of SuperLU and Ng is the group leader of the Scientific Computing Group of the Computation Research Division at Lawrence Berkeley National Laboratory. This algorithm produces the column permutation matrix $\mathbf{P}_c$ and an ordering for column elimination. $\mathbf{P}_c$ is a symmetric permutation of $\mathbf{S}^T\mathbf{S}$ that is determined without explicitly forming $\mathbf{S}^T\mathbf{S}$.

An upper bound on the number of non-zero entries in **L** and **U** are also computed. During numerical factorization the column ordering may be modified. Threshold partial pivoting is used at factorization time to maintain stability. An entry from the pivot column qualifies as a pivot if $\left|a_{ij}\right| \geq 0.1 \max_k \left|a_{kj}\right|$. The sparsest row that meets the criterion is used as the pivot row.

The symmetric strategy pre-orders the columns of **S** using the AMD algorithm. AMD is an approximate minimum degree algorithm developed by Amestoy, Davis, and Duff [3][4]. The AMD algorithm is applied to the pattern $\mathbf{S}+\mathbf{S}^{\mathrm{T}}$. During numerical factorization the column ordering is not modified. Threshold pivoting is used, but a strong preference is given to the diagonal entry. The diagonal entry is used if $\left|a_{jj}\right| \geq 0.001 \cdot \max_k \left|a_{kj}\right|$. Otherwise, a sparse row is selected using the same method as in the unsymmetric strategy.

The 2-by-2 strategy simply applies the symmetric strategy to $\mathbf{P}_2\mathbf{S}$.

The column ordering algorithms produce an elimination tree with each node of the tree corresponding to a dense frontal matrix. A post order traversal of the elimination tree determines the sequence of calculations. Variables are eligible for elimination as soon as they have been fully summed. The analysis phase also determines upper bounds on memory usage, floating point operations, and number of non-zero entries in **LU**.

In the numeric factorization phase, one or more columns of **A** are eliminated in each frontal matrix. The frontal matrices are assembled in dense, 2-dimensional arrays.

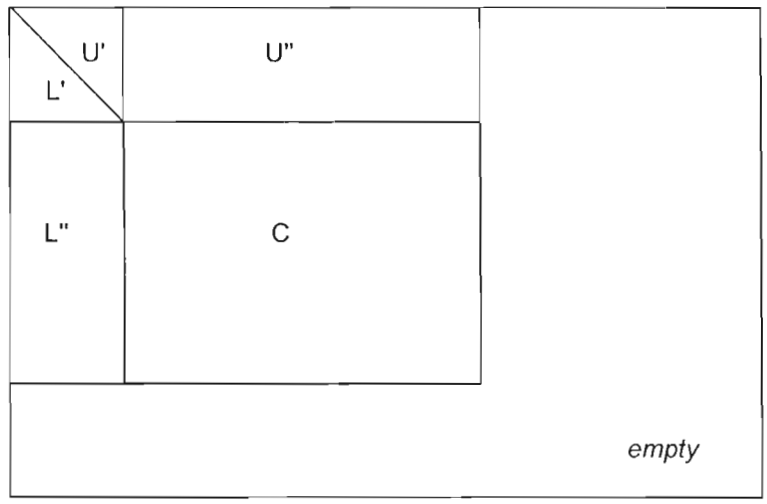

Figure 5. 3. Dense array for assembling a frontal matrix.

The **L** and **U** columns and rows of the frontal matrix are updated as completely summed variables are eliminated. After the completely summed variables are eliminated, the contribution block **C** is updated with BLAS level 3 matrix-matrix multiplication.

$$\mathbf{C} \leftarrow \mathbf{C} - \mathbf{L}''\mathbf{U}''$$

Like SuperLU, UMFPACK estimates the condition number of **A** and computes the componentwise backward error *BERR*. However, UMFPACK does not estimate the forward error *FERR*. UMFPACK also has an option for performing iterative refinement.

UMFPACK consists of a library of 31 user-callable routines. In addition, the AMD ordering method is another library consisting of 4 user-callable routines. Similar to SuperLU's driver routines, there are only a few UMFPACK routines that a user would typically call when solving a system of equations. When solving a system of equations, separate UMFPACK calls are made to

1. Determine the column ordering strategy and perform the initial symbolic factorization
2. Perform the numeric factorization
3. Solve the system of equations from the **LU** factors using forward and backward substitution
4. Free dynamic memory allocated by numeric factorization

5. Free dynamic memory allocated by symbolic factorization.

UMFPACK also has facilities for user specified column ordering.

UMFPACK does not have user specified options for determining which computational steps to skip when solving multiple systems of equations like SuperLU. Instead, similar results are achieved by only calling the routines necessary to perform the needed computations. For example, if a system of equations has multiple right hand sides, the solve routine can be called multiple times without computing the **LU** factors each time. When multiple systems of equations with the same sparsity pattern are solved, the symbolic factorization routine can be called for the first system of equations and the results reused for each subsequent system.

UMFPACK's library supports double precision real numbers and double precision complex numbers. In addition it has support for 32 bit and 64 bit integers. These give a total of four possible versions for many of its routines. Routine names contain a two-letter designation for their data type support. For example, routines names with the prefix "di" operate on double precision real data and use 32 bit integers. Routine names with the prefix "zl" operate on double precision complex data and use 64 bit integers.

The dynamic memory data objects produced by symbolic and numeric factorization routines are opaque to FORTRAN and C programs. Among other things, these objects contain the **L** and **U** factors. While there is no need to directly access the contents of these objects when solving systems of equations, there may be other instances where the contents of the objects are of interest. UMFPACK contains routines for copying **L**, **U**, $\mathbf{P}_r$, $\mathbf{P}_c$, $\mathbf{D}_r$ and other information from the opaque objects to regular arrays.

UMFPACK requires a high performance BLAS to obtain maximum performance. Davis suggests using Goto's BLAS. This is why Goto's BLAS was used in this work. The UMFPACK package includes C implementations of the BLAS routines it uses, so it can be used without BLAS, but overall performance takes a significant hit. See the results section of this work for details. An UMFPACK build time option

determines the library containing the BLAS subprograms to use. SuperLU is similar to UMFPACK in this regard, too.

The matrix **A** is presented to UMFPACK routines in compressed column format. This is the same representation used by MATLAB. The vectors **b** and **x** are represented as dense vectors in 1-dimensional arrays. The UMFPACK library includes various format conversion routines including

1. Triplet representation to compressed column format
2. Compressed column format to triplet representation
3. Transpose of compressed column format to compressed row format

There are a number of user settable parameters that control the operation of UMFPACK. The parameters are stored in a 1-demensional, double precision real array named `CONTROL`. Some of these parameters are defined when the UMFPACK library is built, while the others are specified at run time. The build-time parameters are as follows.

1. `UMFPACK_COMPILED_WITH_BLAS`: True if BLAS is used
2. `UMFPACK_COMPILED_WITH_MATLAB`: True for MATLAB mex functions
3. `UMFPACK_COMPILED_WITH_GETRUSAGE`: 1 if the UMFPACK timer routine bases time measurements on `getrusage` (preferred). Otherwise, time measurements are based on ANSI C `clock` routine.
4. `UMFPACK_COMPILED_IN_DEBUG_MODE`: True if debug mode is enabled.

The run time control parameters are as follows. Default values are listed in parenthesis.

1. `UMFPACK_PRL (1)`: Printing level. 1 is lowest, 6 is highest. Determines the level of detail printed by reporting routines.

2. `UMFPACK_DENSE_ROW (0.2)`: Parameter for defining the number of non-zero entries in a row that constitute a dense row. Dense rows receive special treatment during symbolic and numeric factorization. A row is dense if it contains more than $\max(16, 16\alpha_r\sqrt{n})$ non-zero entries. α_r is the dense row parameter and n is the number of columns.
3. `UMFPACK_DENSE_COL (0.2)` Parameter for defining the number of non-zero entries in a column that constitute a dense column. Dense columns receive special treatment during symbolic and numeric factorization. A column is dense if it contains more than $\max(16, 16\alpha_c\sqrt{n})$ non-zero entries. α_c is the dense column parameter and n is the number of rows.
4. `UMFPACK_PIVOT_TOLERANCE (0.1)`: Threshold for partial pivoting.
5. `UMFPACK_BLOCK_SIZE (32)`: BLAS block size.
6. `UMFPACK_STRATEGY (0=AUTO)`: Strategy for preordering rows and columns.
7. `UMFPACK_ALLOC_INIT (0.7)`: Initial memory allocation as a fraction of estimated peak memory usage.
8. `UMFPACK_IRSTEP (2)`: Maximum number of iterative refinement steps.
9. `UMFPACK_2BY2_TOLLERANCE (0.01)`: Defines large entries for 2-by-2 strategy
10. `UMFPACK_FIXQ (0=AUTO)`: Fix or modify column permutation matrix. UMFPACK uses **Q** to designate $\mathbf{P}_c$.
11. `UMFPACK_AMD_DENSE (10)`: AMD dense row/column parameter.
12. `UMFPACK_SYM_PIVOT_TOLERANCE (0.001)`: Pivot tolerance for diagonal entries with symmetric strategy.
13. `UMFPACK_SCALE (1=SUM)`: Row scaling (none, sum, or max).
14. `UMFPACK_FRONT_ALLOC_INIT (0.5)`: Frontal matrix allocation ratio.
15. `UMFPACK_DROP_TOLERANCE (0)`: Drop tolerance.
16. `UMFPACK_AGGRESSIVE (1=YES)` Aggressive absorption in AMD and COLAMD.

UMFPACK routines are well documented in the 127-page UMFPACK User Guide. The documentation explains what each routine does, what its input and output arguments are, and the influence of control

parameters. Detailed explanations of the control parameters are contained in the documentation of the routines that use them.

Statistics generated by UMFPACK are stored in an information array and are accessible to the user. UMFPACK provides routines for reporting this and other information as summarized below.

1. Printing the status returned by other UMFPACK routines
2. Printing statistics from the information array
3. Printing user defined control settings
4. Printing the symbolic factorization object
5. Printing the numeric factorization object
6. Printing matrices and vectors

Below is a sample of the output produced by UMFPACK when solving a sample problem from the ice sheet model. The demonstration program begins by reading **A** and **b** from a disk file and storing the data in compressed column format. The control parameter settings appear next, followed by the time required to perform symbolic and numeric factorization and solve the system of equations. These are times measured by the demonstration software, not by UMFPACK. Following the timings are information statistics gathered by UMFPACK and printed with the UMFPACK reporting routine. The final section concludes with the standard section of error measures that are included in all test routines in this work.

```
Linear Equations Solver with UMFPACK

 Rodney Jacobs, University of Maine, 2005

 Initialize compressed column data structures
 # Non-zero Elements........:  1629108
 # Columns..................:  24000
 Read and store matrix A
 Read and store righthand side
 Solve Ax=B
UMFSOLVE: initialization
   ncol = 24000
     nz = 1629108
   base = 1
   mode = 1

UMFPACK V4.3.1 (Jan. 11, 2005), Control:

    Matrix entry defined as: double
    Int (generic integer) defined as: int

    0: print level: 3
    1: dense row parameter:    0.2
        "dense" rows have    > max (16, (0.2)*16*sqrt(n_col) entries)
    2: dense column parameter: 0.2
        "dense" columns have > max (16, (0.2)*16*sqrt(n_row) entries)
    3: pivot tolerance: 0.1
    4: block size for dense matrix kernels: 32
    5: strategy: 0 (auto)
    6: initial allocation ratio: 0.7
    7: max iterative refinement steps: 2
    12: 2-by-2 pivot tolerance: 0.01
    13: Q fixed during numerical factorization: 0 (auto)
    14: AMD dense row/col parameter:    10
       "dense" rows/columns have > max (16, (10)*sqrt(n)) entries
        Only used if the AMD ordering is used.
    15: diagonal pivot tolerance: 0.001
        Only used if diagonal pivoting is attempted.
    16: scaling: 1 (divide each row by sum of abs. values in each row)
    17: frontal matrix allocation ratio: 0.5
    18: drop tolerance: 0
    19: AMD and COLAMD aggressive absorption: 1 (yes)

    The following options can only be changed at compile-time:
    8: BLAS library used:  Fortran BLAS.
    9: compiled for ANSI C (uses malloc, free, realloc, and printf)
    10: CPU timer is POSIX times ( ) routine.
    11: compiled for normal operation (debugging disabled)
    computer/operating system: Linux
    size of int: 4 long: 4 Int: 4 pointer: 4 double: 8 Entry: 8 (in bytes)

UMFSOLVE: Symbolic factorization
0.224851 secs for symbolic factorization
UMFSOLVE: Numeric factorization
13.922337 secs for numeric factorization
UMFSOLVE: Solve Ax=b
0.521602 secs to solve
```

Figure 5. 4. Sample UMFPACK output.

```
UMFPACK V4.3.1 (Jan. 11, 2005), Info:
    matrix entry defined as:          double
    Int (generic integer) defined as: int
    BLAS library used:                Fortran BLAS.
    MATLAB:                           no.
    CPU timer:                        POSIX times ( ) routine.
    number of rows in matrix A:       24000
    number of columns in matrix A:    24000
    entries in matrix A:              1629108
    memory usage reported in:         8-byte Units
    size of int:                      4 bytes
    size of long:                     4 bytes
    size of pointer:                  4 bytes
    size of numerical entry:          8 bytes

    strategy used:                    symmetric
    ordering used:                    amd on A+A'
    modify Q during factorization:    no
    prefer diagonal pivoting:         yes
    pivots with zero Markowitz cost:                0
    submatrix S after removing zero-cost pivots:
        number of "dense" rows:                     0
        number of "dense" columns:                  0
        number of empty rows:                       0
        number of empty columns                     0
        submatrix S square and diagonal preserved
    pattern of square submatrix S:
        number rows and columns                     24000
        symmetry of nonzero pattern:                1.000000
        nz in S+S' (excl. diagonal):                1605108
        nz on diagonal of matrix S:                 24000
        fraction of nz on diagonal:                 1.000000
    AMD statistics, for strict diagonal pivoting:
        est. flops for LU factorization:            2.74142e+10
        est. nz in L+U (incl. diagonal):            26015364
        est. largest front (# entries):             3721041
        est. max nz in any column of L:             1929
        number of "dense" rows/columns in S+S':     0
    symbolic factorization defragmentations:        0
    symbolic memory usage (Units):                  3886653
    symbolic memory usage (MBytes):                 29.7
    Symbolic size (Units):                          62613
    Symbolic size (MBytes):                         0
    symbolic factorization CPU time (sec):          0.23
    symbolic factorization wallclock time(sec):     0.23

    matrix scaled: yes (divided each row by sum of abs values in each row)
    minimum sum (abs (rows of A)):              2.99432e+14
    maximum sum (abs (rows of A)):              1.00000e+30

    symbolic/numeric factorization:      upper bound               actual      %
    variable-sized part of Numeric object:
        initial size (Units)                 4402279              4378277    99%
        peak size (Units)                  202200563             33125251    16%
        final size (Units)                 164557605             26116152    16%
    Numeric final size (Units)             164713647             26260194    16%
    Numeric final size (MBytes)               1256.7                200.3    16%
    peak memory usage (Units)              202568424             33493112    17%
    peak memory usage (MBytes)                1545.5                255.5    17%
```

Figure 5.4. Sample UMFPACK output. (Continued)

```
   numeric factorization flops          6.30832e+11          2.74141e+10     4%
   nz in L (incl diagonal)                 69551049             13019663    19%
   nz in U (incl diagonal)                 90728112             13019314    14%
   nz in L+U (incl diagonal)              160255161             26014977    16%
   largest front (# entries)               32895882              3721041    11%
   largest # rows in front                     5391                 1929    36%
   largest # columns in front                  6114                 1929    32%

   initial allocation ratio used:                 0.207
   # of forced updates due to frontal growth:     0
   number of off-diagonal pivots:                 0
   nz in L (incl diagonal), if none dropped       13019663
   nz in U (incl diagonal), if none dropped       13019314
   number of small entries dropped                0
   nonzeros on diagonal of U:                     24000
   min abs. value on diagonal of U:               1.32e-05
   max abs. value on diagonal of U:               1.00e-00
   estimate of reciprocal of condition number:    1.32e-05
   indices in compressed pattern:                 247579
   numerical values stored in Numeric object:     26015222
   numeric factorization defragmentations:        1
   numeric factorization reallocations:           1
   costly numeric factorization reallocations:    0
   numeric factorization CPU time (sec):          12.73
   numeric factorization wallclock time (sec):    13.92
   numeric factorization mflops (CPU time):       2153.50
   numeric factorization mflops (wallclock):      1969.40
   symbolic + numeric CPU time (sec):             12.96
   symbolic + numeric mflops (CPU time):          2115.29
   symbolic + numeric wall clock time (sec):      14.15
   symbolic + numeric mflops (wall clock):        1937.39

   solve flops:                                   1.79497e+08
   iterative refinement steps taken:              1
   iterative refinement steps attempted:          2
   sparse backward error omega1:                  5.19e-16
   sparse backward error omega2:                  4.50e-32
   solve CPU time (sec):                          0.52
   solve wall clock time (sec):                   0.52
   solve mflops (CPU time):                       345.19
   solve mflops (wall clock time):                345.19

   total symbolic + numeric + solve flops:        2.75936e+10
   total symbolic + numeric + solve CPU time:     13.48
   total symbolic + numeric + solve mflops (CPU): 2047.00
   total symbolic+numeric+solve wall clock time:  14.67
   total symbolic+numeric+solve mflops(wallclock) 1880.95
 Error Measures
 ---------------------------------------------
 BERR.......................:   5.18638481E-16
 ||R||infinity..............:   0.00155661441
 ||H||infinity (lower bound):   2.39263608E-31
 ||A||infinity..............:   1.E+30
 ||B||infinity..............:   1.08198E+11
 ||X||infinity..............:   0.0065058553

 Deallocate memory
UMFSOLVE: free Symbolic object
```

Figure 5.4. Sample UMFPACK output. (Continued)

There are several observations to be made.

1. The AMD estimates made for the number of FLOPs, non-zero entries in **L**+**U**, and the largest front during symbolic factorization compare very well with the actual figures from numeric factorization. It is not clear why the upper bounds on these quantities in the numeric factorization phase are so much larger, but the User Guide does say that these estimates can be very loose when using the symmetric strategy or the 2-by-2 strategy.
2. The estimate of the condition number of **A** is similar to the estimate produced by SuperLU.
3. Numeric defragmentation and reorganization occurred once, but is not reported as being costly. It appears that numeric factorization ran efficiently. Considering that actual peak memory usage is only 17% of estimated peak memory usage, one might conclude that initial allocation ratio in the control array should be much less than 0.7 for efficient use of memory. However, the User Guide indicates, and tests corroborate, that when the symmetric strategy is used, the initial allocation is based directly on the number of non-zero entries estimated in **L** and **U** by symbolic factorization and the initial allocation ratio is ignored.
4. Sparse backward error omega1 (*BERR*) is nearly at machine precision. It agrees with the test program's computation of *BERR*, which is to be expected because **A** is scaled by rows only. A definition for sparse backward error omega2 was not found.
5. Infinity norms compare favorably with infinity norms from the SuperLU computations. (The same system of equations is solved in both examples.)
6. The times measured by the demonstration software match the times reported by UMFPACK.

6. Software Interface to the Ice Sheet Model

Using SuperLU and UMFPACK with the ice sheet model involves two interface issues. First, the values of **A**, **b**, and **x** in $\mathbf{Ax} = \mathbf{b}$ must be stored in data structures that are efficient for the modeling software to access and are compatible with SuperLU and UMFPACK. Secondly, the SuperLU and UMFPACK routines must be called from the modeling software. Both of these issues are addressed by this project.

6.1. Data Structures

The ice sheet model places many requirements on the data structures. The data type for entries of **A**, **b**, and **x** is double precision real. **A** is very sparse. There are about 100 non-zero entries per row and tens of thousands of zeros per row. Memory overhead should be kept as low as possible, so a dense two-dimensional array is out of the question. For a problem size of 40x40x5 nodes with 3D velocities such an array would exceed 4 gigabytes of memory. A banded data structure would work for problems without pressure calculations, but would not work for problems with pressure calculations. Even if the banded data structure were used for problems without pressure, the memory for problems with tens of thousands of variables would be very large. For example, a problem with 40x40x5 nodes with 3D velocities, the bandwidth is 1,241, giving a storage size of 238 megabytes. Yet the number of non-zero entries is 1.63 million, requiring only 13 megabytes of memory. Entries of **A** should be addressable by row and column number and access to entries should be as fast as possible. Direct addressing speeds are ideal, but somewhat slower speeds are acceptable. We can expect the ice sheet model to perform 10^6 to 10^8 accesses to entries of **A** in a single time step, so access times should not exceed a few hundred nanoseconds. The row and column addresses of non-zero entries will not be known prior to the first time step, but will remain the same for all time steps after the first. Finally, **b** and **x** are dense vectors.

The double precision real data type is compatible with SuperLU and UMFPACK. Each software package expects **A** to be in compressed column format. Storing **b** and **x** as dense vectors in single dimension arrays is compatible with both software packages.

The compressed column format for **A** meets nearly all the requirements of the ice sheet model. It does not use excessive space to represent sparse matrices and access by row and column numbers is reasonably efficient. The column number defines the lower and upper index values of the row numbers for that column in the row numbers array. The row numbers are stored in ascending order, so a binary search can be done to quickly find the index of a specific row number. The index of the row number is the index of the value of the entry we wish to access in the values array. The only requirement not met by the compressed column format is efficiently populating the data structure during the initial time step. As discussed earlier, compressed column format does not handle the insertion of new values efficiently. Values that are already stored and have higher index values than new values being added must be moved upward to make space for the new entries. This is essentially an insertion sort process with $O(m)$ performance for inserting a single entry, and $O(m^2)$ performance overall.

A simple modification was made to the compressed column data structure that preserved its appearance as compressed column format to SuperLU and UMFPACK, preserved its efficient memory usage and access performance, and dramatically improved element insertion performance in the first time step. The new structure is called modified compressed column format. It consists of four 1-dimensional arrays as illustrated in figure 6.1.

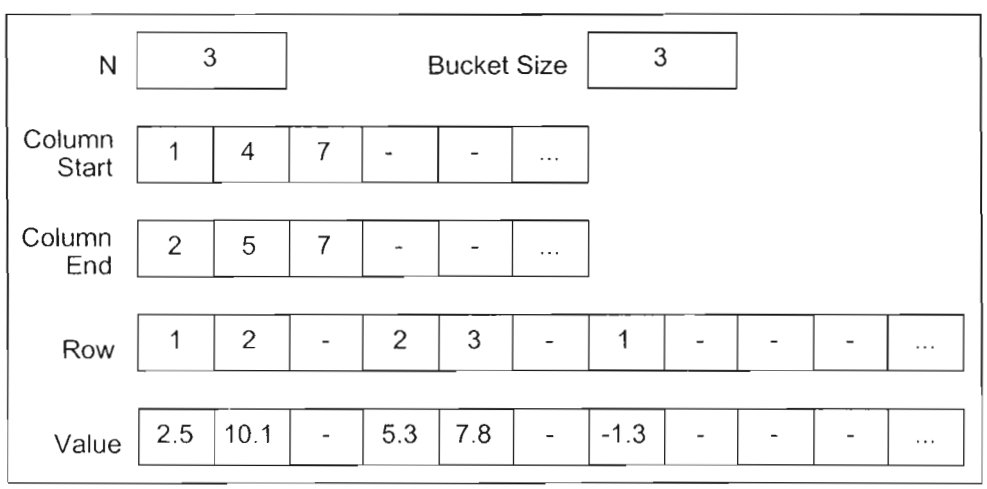

Figure 6. 1. Modified compressed column format.

The Column Start, Row, and Value arrays play the same roles as they do in compressed column format. N is the number of columns stored in the structure. Bucket Size is new. When the data structure is initialized, Bucket Size specifies the number of entries allocated in the Row and Value arrays for each column. The indices of the Column Start and Column End arrays are column numbers. The values in the Column Start array are the index values where the columns begin in the Row and Value arrays. The values in Column End array are the index values where the columns end in the Row and Value arrays. When a new entry is added to an existing column, only the entries in that column's bucket need to move in order to make room for it. For example, if the value 3.2 is added to column 2, row 1, then entries 2 and 3 in Row at index positions 4 and 5 move up one place as well as the values 5.3 and 7.8 in Value. The value 3.2 is stored at index position 4 in Value and 1 is stored at index position 4 in Row. Finally, the Column End value for column 2 is incremented from 5 to 6. As long as there is room in the column bucket, new entries can be added in $O(b)$ time where b is the bucket size. Overall, the process takes $O(mb)$ time where m is the total number of entries.

If a bucket fills up, there are two possible alternatives. The process can simply abort, or the bucket size can be expanded so long as there is sufficient unused space in the Row and Value arrays. The implementation used in this project allows buckets to expand. When a bucket expands, all the entries above that bucket must move up, invoking a performance penalty. Bucket size should be chosen to minimize the number of expansions that occur and the amount of storage space used. If all rows have the same number of non-zero entries and this number is known, then choosing the bucket size is easy. Otherwise, a value must be chosen that balances execution time against the amount of storage space. If the bucket size is at least as large as the largest number of non-zero entries in a row, then no expansions occur and execution time is minimized at the expense of wasted storage space in buckets that are not filled. If bucket size is equal to the number of non-zeros in the row with the least number of non-zeros, then storage space tends to be optimized at the expense of execution time.

Accessing an entry by row and column in the modified compressed column data structure is just as fast as accessing a compressed column data structure. The only difference is that the upper bound for the binary

search is taken from the Column End array. With a regular compressed column data structure, the upper bound of the binary search is the column start index of the next column minus 1.

The modified compressed column data structure is not directly usable by SuperLU and UMFPACK. However, if the free space in each column bucket is removed by moving data at higher positions in the Row and Value arrays down, then the data stored in the Column Start, Row, and Value arrays is the same as the compressed column data structure. Removing the free space is referred to as *squeezing* the data. In the example above, the squeeze operation performs the following moves and changes.

1. Value array moves: 5.3 from index 4 to index 3, 7.8 from index 5 to index 4, –1.3 from index 7 to index 5.
2. Row array moves: 2 from index 4 to index 3, 3 from index 5 to index 4, 1 from index 7 to index 5.
3. Column Start array changes: 4 at index 2 changed to 3, 7 at index 3 changed to 5.
4. Column End array changes: 5 at index 2 changed to 4, 7 at index 3 changed to 5.
5. Column End array change: 6 is inserted at index 4 for compatibility with the compressed column format.

The squeeze operation is performed in the ice sheet model just before SuperLU or UMFPACK is called to solve the system of equations in the first time step. On subsequent time steps of the ice sheet model, the column and row arrays already contain the correct entries. All that needs to be done is set the entries in the Value array to zero before beginning the FEM calculations. There is no need to invoke the squeeze operation in subsequent time steps.

There is one additional detail to handle. SuperLU and UMFPACK are written in C and C uses zero as the starting index of arrays. The ice sheet model is written in FORTRAN and FORTRAN uses one as the starting index of arrays. The modified compressed data structure is created in FORTRAN using base 1 arrays. When SuperLU or UMFPACK is called, the values in the Column Start and Row arrays are decremented by 1, corresponding to C base 0 arrays. After SuperLU and UMFPACK finish using the

arrays, the values in Column Start and Row are incremented by 1, converting them back to FORTRAN base 1 arrays.

6.2. Implementation of Modified Compressed Column Data Structures

A library of FORTRAN routines has been written to maintain the modified compressed column data structure. The source code for these routines can be found in Appendix 1. Data storage for the modified compressed column data structure is allocated in the main program as FORTRAN arrays. An include file, `ccparam.h`, defines `MAXCOL`, the maximum number of columns, and `MAXNZ`, the maximum number of non-zero entries. The include file is included in the main program and each routine in the subroutine library. The dimensioned arrays are as follows.

`ICCPTR(MAXCOL,2)` The first column of this array is the Column Start array in the modified compressed column data structure. The second column is the Column End array. Because FORTRAN arrays are stored in column major order, the Column Start values are contiguous in memory. This is a necessary condition when passing the Column Start values to SuperLU and UMFPACK.

`ICCROW(MAXNZ)` This array is the Row array in the compressed column data structure.

`CCVAL(MAXNZ)` This array is the Value array in the compressed column data structure.

The library routines are summarized below.

`CCINIT (ICCPTR, NCOL, NBKTSZ)`

This routine initializes the compressed column data structure by setting the initial Column Start and Column End values in `ICCPTR` for each column. `NCOL` is the number of columns to allocate and must be less than `MAXCOL`. (The `NCOL+1` entry is used in the compressed column representation to store the last used index of Row and Value + 1.) `NBKTSZ` is the initial size of each bucket. `ICCPTR(i,1)` is initialized

to (i-1)*NBKTSZ+1. ICCPTR(i,2) is initialized to ICCPTR(i,1)-1 to indicate that each column is empty.

CCPUT (IROW, ICOL, VAL, ICCPTR, ICCROW, CCVAL, NCOL)

This routine stores the matrix entry VAL in the modified compressed column data structure. IROW is the row number of the entry and ICOL is the column number of the entry. A binary search of the entries in column ICOL is performed to determine the position where VAL and IROW should be inserted in the CCVAL and ICCROW arrays. If the column's bucket is full, the bucket's size is increased 20% to make room for the new entry and any additional entries in the same column.

CCADD (IROW, ICOL, VAL, ICCPTR, ICCROW, CCVAL, NCOL)

This routine adds the value VAL to the entry for row IROW, column ICOL. If this entry does not yet appear in the modified compressed column data structure, then VAL is added as a new entry to the structure using the same logic as CCPUT. To optimize performance, the common logic is duplicated in this routine to save the overhead of making an additional subroutine call. The logic is fairly minimal, so the amount of duplicate code is quite small. This routine is implemented as a function and it returns the final value of the entry.

CCGET (IROW, ICOL, ICCPTR, ICCROW, CCVAL)

This routine is a function that returns the value of the entry at row IROW, column ICOL. If the specified element is not stored in the structure, then the routine returns zero, as it must be a zero entry.

CCSQZ (ICCPTR, ICCROW, CCVAL, NCOL, NZ)

This routine removes the free space in each bucket by moving data in higher buckets down to fill the space. The pointers in ICCPTR are updated to reflect the new positions of each column in the ICCROW and CCVAL arrays. The number of entries stored in the data structure is returned in the variable NZ.

CCZERO (ICCPTR, NCOL, CCVAL)

This routine sets values in CCVAL to zero at the end of the current time step of the ice sheet model in preparation for the next time step.

In addition, there are two C routines for converting the index pointers in ICCPTR between base 1 arrays and base 0 arrays.

```
void ccbase0(int nz, int ncol, int iccptr[], int iccrow[])
```

This routine decrements the values in iccptr[] and iccrow[]. nz is the number of entries in the data structure and ncol is the number of columns.

```
void ccbase1(int nz, int ncol, int iccptr[], int iccrow[])
```

This routine increments the values in iccptr[] and iccrow[].

These two routines are called as needed from other C routines. They are not called directly from the ice sheet model, but are instead called by the routines that interface the ice sheet model to SuperLU and UMFPACK.

Lastly, there is a C routine for writing the modified compressed column data structure to disk. This routine is not necessary for the ice sheet model, but it provides a convenient method for testing SuperLU and UMFPACK. It can be inserted in the ice sheet model to record the values of **A** and **b** at the point that SuperLU or UMFPACK is called. The recorded values can then be used in repeated test runs of SuperLU and UMFPACK without having to run the entire ice sheet model.

```
int matdump_(int iccptr[], int iccrow[], double cca[], double b[], double x[],
        int *ncol, int *nz, int *base, int *mode, int *debug)
```

The underscore after the routine name makes the name compatible with FORTRAN global symbol naming conventions used by the g77 compiler under Linux, the platform used in this project. A FORTRAN call to MATDUMP is recorded as a call to MATDUMP_ in the global symbol table of the FORTRAN program's

object file. The calling parameters are identical to the calling parameters for invoking SuperLU and UMFPACK from the ice sheet model. The return value is 0 for success or 1 for failure. The following table describes each calling parameter.

Parameter	Description
`iccptr[]`	The Start Column and End Column arrays
`iccrow[]`	The Row array
`Cca[]`	The Values array
`b[]`	The right hand side of the system of equations
`x[]`	The solution vector (not used by this routine)
`*ncol`	The number of columns in A
`*nz`	The number of non-zero entries in the data structure
`*base`	0 => data structure is composed of C base 0 arrays 1 => data structure is composed of FORTRAN base 1 arrays
`*mode`	1=>Dump A and b to disk Other values prevent dumping to disk
`*debug`	0 => Do not display debugging messages 1 => Display debugging messages on stdout

Table 6. 1. Calling parameters for `matdump`.

The output file is written to the current directory with the filename `matrx#` where # is a runtime substitution parameter. It has the value 1 on the first call to the routine and increments by 1 for each subsequent call within the run. A sample output file is shown in Figure 6.2.

```
14 6
1 1 10
3 1 1
2 2 10
4 2 2
1 3 1
3 3 10
5 3 3
2 4 2
4 4 10
6 4 4
3 5 3
5 5 10
4 6 4
6 6 10
1 13
2 28
3 46
4 68
5 59
6 76
```

Figure 6. 2. Sample output of `matdump`.

The first line of output contains the number of entries in **A** and the number of columns in **A**. The next 14 lines contain entries of **A** in triplet notation. The first element of each triplet is the row number, the second element is the column number, and the third element is the entry value. The last 6 lines are entries of the vector **b**. The first element of these entries is the row number and the second element is the value of **b** for the indicated row.

6.3. Performance of Modified Compressed Column Routines

A simple test program, `cctest.f`, was written to test the performance of the modified compressed column routines. The source code for this program can be found at the end of Appendix 1. The program uses data from the ice sheet model written to a disk file by `matdump`. The program performs the following actions.

1. Loads and saves the triplet representation of **A** in arrays.
2. Initializes the modified compressed column data structure with `ccinit`.
3. Adds the entries of **A** to the compressed column data structure using `ccput`.
4. Removes free bucket space by calling `ccsqz`.
5. Gets each entry in the data structure using `ccget`.
6. Zeros all entries in the data structure using cczero.
7. Adds the values of all entries of **A** to the data structure using `ccadd`.

The time required to perform each action is printed to `stdout` by the program. The clock used to measure these times has a precision of 10^{-2} seconds. The observed times divided by the number of entries are shown in Table 2 for a small system of equations and a larger system of equations. The times are in nanoseconds per entry. Both tests were run with `MAXCOL=40,636`, `MAXNZ=4,100,000`, and `NBKTSZ=100`.

Parameter	Small System	Large System
# of Columns (NCOL)	13,500	40,635
# of Entries (NE)	906,048	2,888,361
CCPUT time	106.6	121.2
CCSQZ time	11.0	16.1
CCGET time	66.2	73.8
CCZERO time	7.3	5.7
CCADD time	91.9	107.3

Table 6. 2. Performance of modified compressed column routines.

All performance measurements in this work were made using a 2.66 GHz Intel Pentium 4 processor with 512 KB of cache, a 533 MHz front side bus, and 512 MB of memory. Programs were run under Redhat Linux version 9 and compiled with g77 using –O3 optimization.

Overall, performance is well within the design goals and should be adequate to meet the needs of the ice sheet model. Differences in timings between the two problem sizes are within the limits of the clock's precision. However, cache usage patterns may be causing real increases in timing for the larger problem size. The time for the `ccput` operation is optimistic because the ordering of the input data ensures that new entries are always added to the end of a bucket. The test program was modified to insert entries into buckets in reverse order to see what the worst-case time would be. For the larger problem size, the `ccput` time increased from 121 nanoseconds to 267 nanoseconds per entry, still within design goals. In the ice sheet model new entries will tend to be inserted in increasing order, so the 121 nanoseconds timing is a better indicator of expected performance. `CCADD` time is better than `CCPUT` time in this test because `CCADD` is updating existing entries in the data structure and not adding new entries. The relatively large difference between `ccget` time and `ccadd` time was surprising. The major difference between the two routines is the memory update performed by `ccadd`.

6.4. Computing and Printing Error Measures

The FORTRAN routine `ccberr.f` computes the error measures that have been shown on the previous sample outputs from SuperLU and UMFPACK. The source code for this routine is contained in Appendix 2.

6.5. Procedural Interface To SuperLU

A C routine was written to interface the ice sheet model with the SuperLU expert driver. The source code for this routine and a FORTRAN demonstration program that calls it is contained in Appendix 3. You may want to refer to those listings while reading this description of them.

The function prototype for calling the interface routine is

```
int sluxsolve_ (int iccptr[], int iccrow[], double cca[], double b[],
        double x[], int *ncol, int *nz, int *base, int *mode, int *debug).
```

The calling parameters are the same as the calling parameters for `matdump`. You can refer back to the description of that routine for detail about the parameters. The return value is 0 for success and 1 for failure.

The routine has a number of side effects.

1. Values of user-defined options may change.
2. If $\mathbf{A}$ is equilibrated, then the values of `cca[]` and `b[]` are modified by $\mathbf{D}_r$ and $\mathbf{D}_c$.
3. The solution is stored in `x[]`.
4. Dynamic memory is allocated for various data structures including the column elimination tree, row and column permutation arrays, and row and column scaling arrays. The dynamically allocated memory must be deallocated with an explicit call to `sluxsolve`.

The routine has two modes of operation. The current mode is determined by the value of `mode`. When `mode=1`, the routine solves $\mathbf{Ax} = \mathbf{b}$. Within this mode there are two sub-modes defined by the static variable `pass` stored within the routine. The value of `pass` is initially `0`. In this sub-mode the routine performs the following actions.

1. Initializes user defined options. User defined options are specified by hard coding them into the routine and recompiling the routine. In future work, it would be desirable to set these options under user control through a parameter file stored on disk.
2. User defined options are listed on `stdout` if `debug=1`.
3. Dynamic memory is allocated for data structures.
4. The variable `pass` is set to `1`.
5. The modified compressed column data structure is converted from array base 1 to array base 0.
6. C structures unique to SuperLU are initialized.
7. SuperLU statistics are initialized.
8. The SuperLU expert driver is called to solve $\mathbf{Ax} = \mathbf{b}$.
9. SuperLU statistics are listed on `stdout` if `debug=1`.
10. Dynamic memory allocated for statistics is deallocated.
11. The modified compressed column data structure is converted from array base 0 to array base 1.

When `pass=1`, only actions 5 through 11 are performed. This is done for all time steps of the ice sheet model after the first one.

When `mode=2`, the dynamic data structures allocated in action 3 above are deallocated and `pass` is set to 0. This is the final cleanup call.

As currently coded, this routine does a complete factorization of **A** every time it is called because `options.Fact=DOFACT`. Taking advantage of the constant sparsity pattern of **A** has the potential of reducing execution time of the expert solver by a few percentage points. One way to implement such a change would be to set `options.Fact` to a different mode after calling the expert solver. Since the user-defined options are stored in a static variable, this value would be retained for all subsequent calls.

Program `demo1.f` illustrates the use of the SuperLU interface routine. This program was used to produce the SuperLU test results presented later in this work. It uses values of **A** and **b** computed by the ice sheet model and written to disk files using the `matdump` routine. The program performs the following actions.

1. Reads the first line of a matrix dump to determine the number of non-zero elements and the number of columns in **A**.
2. Calls the `ccinit` routine to initialize the modified compressed column data structure. The bucket size is hard coded in the program. It is set to 81, which is the optimum size for matrices without pressure. The arrays for the data structure are dimensioned within the program based on the parameters specified in `ccparams.h`.
3. The entries of **A** are read from the data file in triplet format and stored in the data structure using calls to `ccput`.
4. The entries of **b** are read from the data file and stored in the array `B`.
5. Values of **A** and **b** are stored in a second set of arrays. The SuperLU expert driver changes the values in the original arrays when it equilibrates **A**. The second copy is used by the routine `ccberr` to compute the error measures based on the original values.
6. The routine `sluxsolve` is called with `mode=1` to solve the system of equations.
7. The solution vector is printed to `stdout`. This is optional
8. The routine `ccberr` is called to compute and print the error measures.
9. The routine `sluxsolve` is called with `mode=2` to free dynamically allocated memory.

6.6. Procedural Interface to UMFPACK

A C routine was also written to interface the ice sheet model with UMFPACK. The source code for this routine and a FORTRAN demonstration program that calls it is contained in Appendix 4. Again, you may want to refer to those listings while reading this description of them.

The function prototype for calling the interface routine is

```
int umfsolve_ (int iccptr[], int iccrow[], double cca[], double b[],
       double, x[], int *ncol, int *nz, int *base, int *mode, int *debug)
```

This routine has a number of side effects, too.

1. Values of user-defined options may be changed and stored.
2. The solution is stored in `x[]`.
3. Results of symbolic factorization are stored for reuse on subsequent calls when the sparsity pattern of **A** is the same on those calls.

The routine has two modes of operation like the SuperLU interface. The current mode is determined by the value of `mode`. When `mode=1`, the routine solves $\mathbf{Ax} = \mathbf{b}$. Within this mode, there are two sub-modes defined by the start variable `pass`. The value of `pass` is initially `0`. In this sub-mode the routine performs the following actions.

1. Initializes user defined options. User defined options are specified by hard coding them into the routine and recompiling the routine. In future work, it would be desirable to set these options under user control through a parameter file stored on disk.
2. The modified compressed column data structure is converted from array base 1 to array base 0.
3. Symbolic factorization is performed to determine column ordering based on the sparsity pattern of **A**.

4. If `pass` is set to `0`, it is set to `1`. Otherwise, the modified compressed column data structure is converted from array base 1 to array base 0.
5. Numeric factorization is performed to compute **L** and **U**.
6. Forward and backward substitution is used to solve for **x**.
7. Memory allocated for the results of numeric factorization is freed.
8. UMFPACK statistics are listed on stdout if `debug=1`.
9. The modified compressed column data structure is converted from array base 0 to array base 1.

When `pass=1`, only actions 4 through 9 above are performed. This is done for all time steps of the ice sheet model except the first one.

When mode=2, dynamic memory allocated for the results of symbolic factorization is freed and `pass` is reset to `0`. This is the final cleanup call.

Program `demo.f` illustrates the use of the UMFPACK interface routine. It is identical to the SuperLU demonstration program `demo1.f` except that it does not store copies of **A** and **b** before calling UMFPACK. UMFPACK does not modify the original values of **A** and **b**, so there is no need to save copies of them for calling the error measures routine `ccberr`. The initial design goal had been that the same demo program would work for both methods. While either demo program will in fact work properly with SuperLU and UMFPACK, the minor differences between them allow us to compare the output of `ccberr` for both methods.

7. Establishing a Basis For Performance Measures

Typically, the performance of new methods for solving systems of linear equations is evaluated by comparing the new methods to established methods. The authors of SuperLU and UMFPACK have done this. However, we lack experience with the prior systems these authors discuss, so the comparisons are not particularly revealing to us. On the other hand, banded Gaussian elimination (BGE) has been implemented in the 3D ice sheet problem without pressure and does provide a basis for evaluating SuperLU and UMFPACK. In addition, the results reported by these authors show that performance characteristics are not uniform across all sparse matrices. Therefore, comparing SuperLU, UMFPACK, and BGE with sparse matrices from the ice sheet model should give a good indication of how the methods will perform in the model.

Initially, a simple BGE program was written in FORTRAN. The source code for this program is contained in Appendix 5. It uses data from the ice sheet model written to a disk file by `matdump`. The program performs the following actions.

1. The data file is read to determine the lower and upper bandwidths of **A**.
2. With the bandwidth known, the data file is reread and the entries of **A** are stored in a 2-dimensional array using the banded matrix storage technique discussed earlier. Gaussian elimination is performed in place, so a second copy of **A** is also stored for computing error measures at the end of the program.
3. Entries of **b** are read from the data file and stored. A second copy is also stored for computing error measures.
4. **A** is reduced to upper triangular form using Gaussian elimination without row interchanges.
5. The results of Gaussian elimination are combined with backward substitution to compute **x**.
6. **x** is printed. This is optional.
7. Timings and floating point operation counts are printed.
8. Backward error is computed and printed.

Sample results from this program are shown in the next figure.

```
Banded Gaussian Elimination (BG0)
   Assume Constant Upper Bandwidth
Rodney Jacobs, University of Maine, 2005

Bandwidth determination
# Non-zero Elements............:  393588
# Rows........................:  6000
Lower Bandwidth..............:  320
Upper Bandwidth..............:  320
Read and store matrix A
Read and store righthand side
Reduce A to upper triangular form
Sove for x using backward substitution
Triangular Reduction Time (sec).:   4.56999969
Backward Substitution Time (sec):   0.0100002289
Total Solve Time (sec)..........:   4.57999992
Triangular Reduction FLOPS......:  1201535520
Backward Substitution FLOPS.....:  3743280
Total Solve FLOPS...............:  1205278800
BERR........................:   1.84623029E-15
```

Figure 7. 1. Output from `bg0.f`.

Interestingly, *BERR* is near machine precision indicating a stable set of calculations without partial pivoting or iterative refinement. This tends to be characteristic of the ice sheet equations without pressure. The matrices are not diagonally dominant, but they are symmetric and may be positive definite or nearly so.

Additional refinements were made to `bg0.f` to test the following issues.

1. Does counting of the number of FLOPS increase the execution time significantly? Judging for the code it is not expected to, but the counting operations can be temporarily removed and the program rerun to see for sure.
2. For optimum cache hits, the array for storing **A** in banded format is arranged so that entries for each row are contiguous in memory. What is the performance cost if columns of entries are stored contiguously in memory instead of rows of entries?

3. If the 2-D matrix for storing **A** is dimensioned for a larger bandwidth than the bandwidth of **A**, what happens to performance? We can expect lower cache hit rates because the data will be more spread out in memory.
4. The rectangular arrangement of the FEM nodes in the ice sheet model results in some rows having a smaller upper or lower bandwidth than others. Are there significant performance benefits to storing and using this information to reduce the total number of floating point operations performed?

A 20x20x5 node ice sheet matrix for 3D velocities was used to obtain the following answers to these questions.

1. Counting FLOPs increased execution time by 0.2%, certainly not very significant. For larger problems, this ratio can be expected to be even smaller.
2. Optimizing for cache hits is crucial to good performance. With entries of **A** stored contiguously by column instead of by row, execution time increased by a factor of 6.5.
3. The 20x20x5 node problem has a bandwidth of 641. The parameter `MAXBW` in `bg0.f` determines the number of matrix elements allocated for each row of **A**. Runtimes were measured for `MAXBW` = 641, 941, and 1241. The respective runtimes were 4.18 seconds, 4.32 seconds, and 4.43 seconds, an overall 6.0% increase in runtime.
4. Tracking individual row bandwidths in order to reduce the number of floating point operations decreased runtime by 4.4%.

In order to investigate partial pivoting, scaling, and iterative refinement with banded matrices, a library of banded matrix routines was written to support both banded Gaussian elimination and banded **LU** factorization. The expanded BGE routines are listed in Appendix 6 and the banded **LU** factorization routines are listed in Appendix 7. In keeping with the designs of SuperLU and UMFPACK, control and information arrays were defined for specifying user settable parameters and obtaining statistics from the

processes. These arrays are defined in `bandparam.h` in Appendix 6. The contents of the arrays are summarized in the following two tables.

Control	Description
`BCTL(1)`	Reporting 0 = Display severe errors only 1 = Display warning messages and severe errors 2 = Display progress and warning messages and severe errors 3 = Display statistics and all other messages
`BCTL(2)`	Print solution vector 0 = No 1 = Yes
`BCTL(3)`	Scale matrix before reducing or factoring it 0 = Do not scale 1 = Scale each row by sum of absolute values of coefficients and right hand side value in that row
`BCTL(4)`	Matrix permutations 0 = Do not perform any permutations 1 = Partial pivoting by row interchanges based on BCTL(5)
`BCTL(5)`	Partial pivoting threshold (BGE only) Value must be between 0 and 1
`BCTL(6)`	Iterative refinement (LU factorization only) Max number of iterative refinement steps. Stop iterating sooner if there is no improvement in x. Increasing *BERR* is used as the indicator ofno improvement in x. If zero, do not perform iterative refinement. If less than zero, take the absolute value of this number and Perform exactly this number of iterative refinement steps.
`BCTL(7)`	Print solution vector after performing iterative refinement 0 = No 1 = Yes

Table 7. 1. Banded matrix control array.

Element	Description
BINFO(1)	Number of non-zero elements in matrix
BINFO(2)	Number of rows in matrix
BINFO(3)	Lower bandwidth of matrix
BINFO(4)	Upper bandwidth of matrix
BINFO(5)	BGE reduction FLOPs
BINFO(6)	BGE reduction wall clock time (seconds)
BINFO(7)	BGE backward substitution FLOPs
BINFO(8)	BGE backward substitution wall clock time (seconds)
BINFO(9)	LU factorization FLOPs
BINFO(10)	LU factorization wall clock time (seconds)
BINFO(11)	LU forward + backward substitution FLOPs
BINFO(12)	LU forward + backward substitution wall clock time
BINFO(13)	LU factorization: Number of non-zeros in L excluding diagonal
BINFO(14)	LU factorization: Number of non-zeros on diagonal of U
BINFO(15)	LU factorization: Number of non-zeros in U excluding diagonal
BINFO(16)	BERR (Componentwise backward error)
BINFO(17)	Infinity norm of residual vector
BINFO(18)	Infinity norm of H (lower bound) (H is perturbation of A for which the computed solution is the exact solution of (A + H) x = b)
BINFO(19)	Infinity norm of A
BINFO(20)	Infinity norm of b
BINFO(21)	Infinity norm of x
BINFO(22)	Number of iterative refinement steps performed
BINFO(23)	Number of row permutations performed

Table 7. 2. Banded matrix information array.

The following table summarizes each of the banded Gaussian elimination routines. Each of these routines, except bge, is used with banded **LU** factorization as well.

Routine	Description
binit	Initialize BCTL and BINFO arrays with default values.
buser	Let the user specify BCTL values interactively.
bload	Read a matdump file. Store A as a banded matrix and b in a 1-D array.
bscale	Scale rows of A.
bcopy	Copy A and b to a second set of arrays.
bge	Solve Ax = b using banded Gaussian elimination.
berror	Compute error measures.

Table 7. 3. Banded Gaussian elimination routines.

Each of these routines is straightforward with bge being the most complicated. bge optimizes the amount of calculation by considering the bandwidth of each row of **A**. It does partial threshold pivoting and checks for zero pivot values.

The demonstration program for the BGE library is `bgauss.f`. Appendix 6 contains a listing of the program. In summary, the program performs the following actions.

1. The control and information arrays are initialized by calling `binit`.
2. The user is given the opportunity to interactively specify control parameter values by calling `buser`.
3. The matrix **A** and the right hand side **b** are read from disk by calling `bload`.
4. **A** is scaled by calling `bscale`.
5. Copies of **A** and **b** are saved for computing error measures by calling `bcopy`.
6. Gaussian elimination and backward substitution are used to solve the system of equations by calling `bge`.
7. The solution is printed if the option to do so is selected in the control array.
8. Error measures are computed and displayed by call `berror`.

Sample output produced by this program is shown in Figure 7.2.

```
Banded Gaussian Elimination

Rodney Jacobs, University of Maine, 2005

Control Settings
-----------------------------------------------------------------------
 1. Display all messages plus statistics
 2. Do not print solution vector
 3. Do not scale matrix
 4. Do not perform any matrix permutations
 5. Partial pivoting threshold =  0.1
 6. Perform at most  3. iterative refinement steps with LU factorization
 7. Do not print solution vector after each iterative refinement step

Enter # to change or 0:
BLOAD: Bandwidth determination
BLOAD: # Non-zero Elements.:  393588
BLOAD: # Rows..............:  6000
BLOAD: Lower Bandwidth.....:  320
BLOAD: Upper Bandwidth.....:  320
BLOAD: MAXROW..............:  24000
BLOAD: MAXBW...............:  1241
BLOAD: Initialize arrays
BLOAD: Read and store matrix A
BLOAD: Read and store righthand side
BCOPY: Copy arrays representing a banded matrix
BGE: Reduce A to upper triangular form
BGE: Solve for x using backward substitution
Gaussian Elimination Statistics
--------------------------------------------
No row interchanges
Reduction Time (sec).......:   4.4000001
Substitution Time (sec)....:   0.00999927521
Total Solve Time (sec).....:   4.40999937
Reduction FLOPS............:   1.14919848E+09
Substitution FLOPS.........:   3718962.
Total Solve FLOPS..........:   1.15291744E+09
BERROR: Compute error measures
Error Measures
--------------------------------------------
BERR.......................:   1.84623029E-15
||R||infinity..............:   0.0195611205
||H||infinity (lower bound):   1.90147725E-30
||A||infinity..............:   1.E+30
||B||infinity..............:   4.55868E+11
||X||infinity..............:   0.0102873282
```

Figure 7. 2. Sample output of `bgauss.f`.

The banded **LU** factorization routines are summarized in the following table.

Routine	Description
`blufac`	Factor A into LU.
`blusolve`	Solve LUx = b using forward and backward substitution.
`blustats`	Display statistics for the factorization of A and solving of LUx = b.
`blurefine`	Iteratively refine the solution x.

Table 7. 4. Banded LU factorization routines.

The demonstration program for the banded **LU** factorization library is `blu.f`. Appendix 7 contains a listing of the program. In summary, the program performs the following actions.

1. The control and information arrays are initialized by calling `binit`.
2. The user is given the opportunity to interactively specify control parameter values by calling `buser`.
3. The matrix **A** and the right hand side **b** are read from disk by calling `bload`.
4. **A** is scaled by calling `bscale`.
5. Copies of **A** and **b** are saved for performing iterative refinement and computing error measures by calling `bcopy`. Iterative refinement is performed using the scaled arrays. Also, error measures are computed using the scaled arrays.
6. **A** is factored by calling `blufac`.
7. **LUx** = **b** is solved for **x** by calling `blusolve`.
8. Statistics for performing the factorization and solving for **x** are displayed by calling `blustats`.
9. Error measures are computed and displayed by call `berror`.
10. Iterative refinement is performed by calling `blurefine`.

Sample output produced by this program is shown in Figure 7.3.

```
Banded LU Factorization with Iterative Refinement
 Rodney Jacobs, University of Maine, 2005


 Control Settings
 ---------------------------------------------------------------------------
  1. Display all messages plus statistics
  2. Do not print solution vector
  3. Do not scale matrix
  4. Do not perform any matrix permutations
  5. Partial pivoting threshold =  0.1
  6. Do not perform iterative refinement with LU factorization
  7. Do not print solution vector after each iterative refinement step

 Enter # to change or 0:
 BCTL(6)=  0. : Perform iterative refinement (LU solutions only)
   0 = No iterative refinement
  >0 = Number of refinement steps.  Stop early if no furthre improvement
  <0 = -1 * exact number of refinement steps
 Enter value:
 ctl=  5.

 Control Settings
 ---------------------------------------------------------------------------
  1. Display all messages plus statistics
  2. Do not print solution vector
  3. Do not scale matrix
  4. Do not perform any matrix permutations
  5. Partial pivoting threshold =  0.1
  6. Perform at most  5. iterative refinement steps with LU factorization
  7. Do not print solution vector after each iterative refinement step

 Enter # to change or 0:
 BLOAD: Bandwidth determination
 BLOAD: # Non-zero Elements.:  393588
 BLOAD: # Rows..............:  6000
 BLOAD: Lower Bandwidth.....:  320
 BLOAD: Upper Bandwidth.....:  320
 BLOAD: MAXROW..............:  24000
 BLOAD: MAXBW...............:  1241
 BLOAD: Initialize arrays
 BLOAD: Read and store matrix A
 BLOAD: Read and store righthand side
 BCOPY: Copy arrays representing a banded matrix
 BLUFAC: Factor A to LU form
 BLUSOLVE: Solve Ly=b using forward substitution
 BLUSOLVE: Solve Ux=y using backward substitution
 BLUSTATS: Count non-zero entries in L and U
 LU Statistics
 ---------------------------------------------
 Non-zeros in L w/diagonal..:   1821900.
 Non-zeros in U w/diagonal..:   1821900.
 Non-zeros in L+U...........:   3637800.
 LU Factorization Time (sec):   4.4000001
 Substitution Time (sec)....:   0.0199995041
 Total Solve Time (sec).....:   4.4199996
 LU Factorization FLOPS.....:   1.14556668E+09
 Substitution FLOPS.........:   7350762.
 Total Solve FLOPS..........:   1.15291744E+09
```

Figure 7. 3. Sample output of `blu.f`.

```
BERROR: Compute error measures
Error Measures
---------------------------------------------
BERR.......................:   7.46892707E-16
||R||infinity..............:   0.0202770683
||H||infinity (lower bound):   1.97107236E-30
||A||infinity..............:   1.E+30
||B||infinity..............:   4.55868E+11
||X||infinity..............:   0.0102873282
BREFINE: Iterative refinement
BLUSOLVE: Solve Ly=b using forward substitution
BLUSOLVE: Solve Ux=y using backward substitution
BERROR: Compute error measures
Error Measures
---------------------------------------------
BERR.......................:   5.45366524E-17
||R||infinity..............:   0.00178424106
||H||infinity (lower bound):   1.73440667E-31
||A||infinity..............:   1.E+30
||B||infinity..............:   4.55868E+11
||X||infinity..............:   0.0102873282
BLUSOLVE: Solve Ly=b using forward substitution
BLUSOLVE: Solve Ux=y using backward substitution
BERROR: Compute error measures
Error Measures
---------------------------------------------
BERR.......................:   5.60270716E-17
||R||infinity..............:   0.00210852362
||H||infinity (lower bound):   2.04963192E-31
||A||infinity..............:   1.E+30
||B||infinity..............:   4.55868E+11
||X||infinity..............:   0.0102873282
BREFINE: No further improvement
BREFINE:  1 refinement step(s) performed
```

Figure 7.3. Sample output of blu.f. (Continued)

The same system of equations is solved in the BGE and banded **LU** examples. Some initial observations can be made from the outputs of the two programs.

1. The run times of the two programs, excluding iterative refinement in `blu.f`, are essentially identical.
2. The numbers of floating point operations performed by the two programs, excluding iterative refinement in `blu.f`, are identical. This is expected. The two methods order the calculations a little differently, but they perform the same calculations.
3. The infinity norm of **x** agrees to 10 places between the two solutions.
4. One step of iterative refinement reduced *BERR* and the infinity norm of the residual by more than a factor of 10. This did not change the infinity norm of **x** to 10 places.

8. Testing and Benchmarking SuperLU and UMFPACK

8.1. Verification Testing

As an initial verification test, all programs were required to solve

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 8 & 0 & 0 & 0 \\ 0 & 0 & 9 & 10 & 11 & 0 & 0 \\ 0 & 0 & 0 & 12 & 13 & 14 & 0 \\ 0 & 0 & 0 & 0 & 15 & 16 & 17 \\ 0 & 0 & 0 & 0 & 0 & 18 & 19 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 5 \\ 26 \\ 65 \\ 122 \\ 197 \\ 290 \\ 241 \end{pmatrix}, \tag{8.1}$$

which has the solution

$$\mathbf{x} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix}^{\mathrm{T}}. \tag{8.2}$$

All programs passed this test.

Next, the poorly scaled problem from Chapter 2,

$$\begin{pmatrix} 0.001 & 2.42 \\ 1.00 & 1.58 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 5.20 \\ 4.57 \end{pmatrix} \tag{8.3}$$

was solved by each program to see how error measures compared. The following error measurements were observed.

Program & Options	BERR	||R||infinity	κ (estimate)
BGAUSS	3.71E-14	3.39E-13	
BGAUSS w/partial pivoting	3.26E-17	3.39E-16	
BLU	3.71E-14	3.39E-13	
BLU w/iterative refinement	3.26E-17	3.39E-16	
SuperLU	2.32E-17	2.42E-16	4.27
UMFPACK auto	3.71E-14	3.39E-13	2.42E+06
UMFPACK auto w/iterative refine	8.01E-17	8.33E-16	2.42E+06
UMFPACK unsymmetric	2.64E-17	2.41E-16	2.42

Table 8. 1. Test results for solutions of Equation 8.3.

Machine precision for the computer running these tests is approximately 10^{-16}. With partial pivoting, BGAUSS is achieving machine precision. Without pivoting, *BERR* and $\|\mathbf{R}\|_\infty$ for BGAUSS are 10^3 times larger, which is reasonable given the pivot growth of 10^3. BLU gives the same error measures as BGAUSS without pivoting. As discussed earlier, banded **LU** factorization is not compatible with partial pivoting. However, iterative refinement is possible. With iterative refinement, BLU produces the same error measures as BGAUSS with partial pivoting.

SuperLU without iterative refinement or additional options produces results comparable to BGAUSS with partial pivoting. This indicates that SuperLU is automatically controlling pivot growth and, in fact, SuperLU reports pivot growth to be 1.0.

UMFPACK is a little more complicated. By default, UMFPACK automatically chooses to solve the system using its symmetric strategy and does not perform row interchanges. This results in high error measures comparable to BGAUSS without partial pivoting. If iterative refinement is invoked, error measures are brought in line with BLU with iterative refinement. UMFPACK normally does iterative refinement by default. Manually selecting UMFPACK unsymmetric mode and turning off iterative refinement forces partial pivoting and reduces error measures to levels comparable to SuperLU, BGAUSS with partial pivoting, and BLU with iterative refinement.

The condition number for **A** is easily computed. It is $\kappa_\infty = \|\mathbf{A}\|_\infty \|\mathbf{A}^{-1}\|_\infty = 4.26$. SuperLU's estimate of the condition number is 4.27. UMFPACK's estimate is anomalous.

We can also look at results for the solution of Equation 8.1 in Table 8.2. (The column labeled "FLOPs" is floating point operations. The label "FLOPS" is reserved for floating point operations per second.)

Program & Options	BERR	‖R‖infinity	κ (estimate)	FLOPs
BGAUSS	4.44E-17	1.24E-14	-	49
BGAUSS w/partial pivoting	0	0	-	49
BLU	2.73E-17	5.33E-15	-	49
BLU w/iterative refinement	0	0	-	Not available
SuperLU	0	0	152	213
SuperLU w/scaling	0	0	152	213
UMFPACK auto	1.73E-15	6.80E-13	1.91E+04	49
UMFPACK auto w/iterative refine	5.47E-17	7.11E-15	1.91E+04	328
UMFPACK auto w/iterative refine and scaling	5.82E-17	1.60E-14	2.35E+05	384
UMFPACK unsymmetric	5.47E-17	2.75E-14	31.6	49
UMFPACK unsym w/iterative refine	5.47E-17	2.75E-14	31.6	160
UMFPACK unsym w/iterative refine and scaling	5.12E-17	1.15E-14	6.29	195

Table 8. 2. Test results for solution of Equation 8.1.

BGAUSS and BLU produce componentwise backward error at machine precision without partial pivoting or iterative refinement. Partial pivoting for BGAUSS and iterative refinement for BLU brings the error measures to zero.

SuperLU takes the error measures to zero directly, but does 4 times more floating point operations than BGAUSS and BLU without iterative refinement. It takes approximately 30 FLOPS for BLU to perform one cycle of iterative refinement on this system of equations, giving a total of 79 FLOPS for BLU to run with iterative refinement. Still, these are 2.7 times fewer FLOPs than SuperLU. SuperLU equilibration was enabled to see if the condition number estimate would change. It did not. This seems strange. The value of the condition number is observed to change when SuperLU's scaling is enabled for other matrices. An independent calculation of the condition number of this matrix gives $\kappa_\infty = 98.4$, so an estimate of 152 without scaling is reasonable. SuperLU does not count floating point operations for equilibrating the matrix, computing the condition number, or performing iterative refinement.

UMFPACK's automatic strategy selection chose the symmetric strategy again. UMFPACK still has problems with the condition number estimate for the symmetric strategy, but the unsymmetric strategy gives a number that is in the ballpark. Iterative refinement and scaling are UMFPACK's default settings. With these settings the error measures are comparable to BGAUSS and BLU without partial pivoting or iterative refinement. However, these settings cause the number of floating point operations to drastically exceed the counts for BGAUSS and BLU. Since SuperLU is not counting floating point operations for scaling and iterative refinement, it is hard to say how UMFPACK and SuperLU compare.

UMFPACK's unsymmetric strategy works better for this matrix. Error measures are low without iterative refinement or scaling and floating point operation counts are lower. UMFPACK's row scaling reduced the condition number for the unsymmetric strategy, which was expected. However, the reliability of UMFPACK's estimate is questionable. An independent computation of the condition number could be done for the scaled matrix to check the numbers.

8.2. Test Matrices

Two series of matrices were generated from the ice sheet model for benchmarking UMFPACK and SuperLU as well as BGAUSS and BLU. One series does not include pressure and the other series does.

Name	Rows	Non-Zeros	Band-witdth	Diag. Non-Zeros	Struct. Sym-metry	Value Sym-metry	Diagonal Dominance
m3d.10x10x5	1,500	91,728	341	1,500	1.0	1.0	0.2
m3d.15x15x5	3,375	216,333	491	3,375	1.0	1.0	0.2
m3d.20x20x5	6,000	393,588	641	6,000	1.0	1.0	0.2
m3d.30x30x5	13,500	906,048	941	13,500	1.0	1.0	0.2
m3d.40x40x5	24,000	1,629,108	1,241	24,000	1.0	1.0	0.2
m3dp.10x10x5	1,824	107,280	N/A	1,500	1.0	1.0	0.164
m3dp.15x15x5	4,159	253,965	N/A	3,375	1.0	1.0	0.162
m3dp.20x20x5	7,444	462,900	N/A	6,000	1.0	1.0	0.161
m3dp.30x30x5	16,864	1,067,520	N/A	13,500	1.0	1.0	0.160

Table 8. 3. Test matrices from the ice sheet model.

Matrix names beginning with "m3d" are banded matrices for 3-D velocities without pressure. Matrix names beginning with "m3dp" are non-banded matrices for 3-D velocities with pressure. The remainder of

the matrix name is the number of FEM nodes in each dimension. An entry a_{ij} is structurally symmetric if a_{ji} is also non-zero. Structural symmetry is the ratio of the number of structurally symmetric entries to the total number of non-zero entries. An entry a_{ij} is value symmetric if $a_{ij} = a_{ji}$. Value symmetry is the ratio of the number of value symmetric entries to the total number of non-zero entries. The test matrices are structurally symmetric as well as value symmetric. Diagonal dominance is the ratio of the number of columns that are diagonally dominant to the total number of columns.

8.3. Initial Tests of BGAUSS

BGAUSS is only applicable for banded matrices. The initial tests look at error measures and number of floating point operations as a function of scaling and partial pivoting. Matrix m3d.20x20x5 was used to obtain these results.

Scale	Pivot Threshold	Row Interchanges	BERR	‖R‖infinity	FLOPs
No	0.0	0	1.85E-15	1.95E-02	1.15E+09
No	0.1	2,941	2.78E-13	5.51E-03	2.22E+09
Yes	0.0	0	1.92E-15	2.63E-02	1.15E+09
Yes	0.2	0	1.92E-15	2.63E-02	1.15E+09
Yes	0.3	23	1.77E-15	4.10E-02	1.15E+09
Yes	0.4	38	1.97E-15	2.86E-02	1.15E+09
Yes	0.5	76	1.71E-15	3.29E-02	1.22E+09
Yes	0.6	1649	5.16E-14	4.57E-02	2.13E+09

Table 8. 4. Initial BGAUSS tests with m3d.20x20x5.

Without scaling, nearly half the rows are interchanged, causing *BERR* to increase by a factor of 100 and the number of floating point operations to increase by 93%, although $\|\mathbf{R}\|_\infty$ decreased by a factor of 4. With scaling, the pivot threshold can be adjusted to control the number of row interchanges. However, there is no threshold value that produces significantly better results. Overall, the calculations are stable without row interchanges even though the matrix is not diagonally dominant.

8.4. Initial Tests of BLU

Like BGAUSS, BLU is only applicable for matrices without pressure. BLU allows iterative refinement and scaling to be performed. Matrix m3d.20x20x5 was used to obtain the following results.

Scale	Iterative Refinement	BERR	||R||infinity	FLOPs
No	No	7.47E-16	2.03E-02	1.15E+09
No	Yes	5.45E-17	1.78E-03	1.15E+09
Yes	No	7.32E-16	7.23E-20	1.15E+09
Yes	Yes	5.04E-17	7.57E-20	1.15E+09

Table 8. 5. Initial BLU tests with m3d.20x20x5.

Iterative refinement improved *BERR* and $\|\mathbf{R}\|_\infty$ with negligible additional cost. Error measures with scaling are more difficult to interpret. Because iterative refinement is based on the scaled system of equations, the error measures are also based on the scaled equations. The norm of the residual using the scaled system of equations is much better than the residual of the unscaled equations. This is because the residual is the difference of large numbers for the unscaled system, but it is the difference of small numbers for the scaled system. BGAUSS demonstrated that scaling probably did little to change the value of **x**. This is inferred from the fact that the norm of the residual in BGAUSS is always calculated from the unscaled equations, and the fact that the value of **x** computed without scaling has a residual norm that is comparable to the residual norm when **x** is computed with scaling.

8.5. Initial Tests of SuperLU

Initial tests of SuperLU with the m3d.20x20x5 matrix produced the following results.

Column Permutation	Scale	I.R.	BERR	||R||infinity	FLOPs	Time (secs)	Memory (MB)
Natural	No	No	5.50E-13	1.90E-02	1.97E+09	3.62	52.3
Natural	Yes	No	1.85E-15	1.42E-02	1.14E+09	1.95	37.5
Natural	Yes	Yes	4.51E-16	5.14E-03	1.14E+09	2.23	37.5
A^T+A	No	No	1.86E-12	1.07E-01	7.43E+10	104.3	241.2
A^T+A	Yes	No	1.26E-15	8.23E-03	6.50E+08	0.960	23.6
A^T+A	Yes	Yes	4.62E-16	6.64E-03	6.50E+08	1.14	23.6
A^TA	No	No	4.01E-13	1.16E-02	2.36E+09	3.13	48.9
A^TA	Yes	No	1.51E-15	1.21E-02	1.32E+09	1.91	33.8
A^TA	Yes	Yes	4.89E-16	4.44E-03	1.32E+09	2.15	33.8
COLAMD	No	No	5.65E-13	1.36E-02	2.05E+09	3.81	51.0
COLAMD	Yes	No	1.35E-14	1.33E-02	1.15E+09	2.08	35.5
COLAMD	Yes	Yes	4.68E-16	5.80E-03	1.15E+09	2.34	35.5

Table 8. 6. Initial SuperLU tests with m3d.20x20x5.

With SuperLU we need to evaluate the column permutation method as well as the effects of scaling and iterative refinement. The most outstanding result is that scaling has a significant impact on all four column permutation methods. SuperLU's equilibration reduces fill-in, thus reducing the number of floating point operations, the runtime, and the amount of memory used. The effects of iterative refinement are much less dramatic, and are in line with improvements seen in BGAUSS and BLU. The multiple minimum degree algorithm operating on the structure of A^T+A is particularly effective at solving this system of equations when scaling is used. Without scaling, this method was very ineffective. The amount of memory and the number of floating point operations far exceeded the other tests. Six memory expansions were required and the runtime was abysmal.

It is also interesting to look at the condition number estimates. Without scaling, $\kappa = 2.55 \cdot 10^{21}$. If the condition number is truly this large, then there may be problems with the 3-D ice sheet model. From Chapter 2 we have

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa \frac{\|\delta \mathbf{b}\|}{\|\mathbf{b}\|} \tag{8.4}$$

and

$$\frac{\|\delta \mathbf{x}\|}{\|\mathbf{x} + \delta \mathbf{x}\|} \leq \kappa \frac{\|\delta \mathbf{A}\|}{\|\mathbf{A}\|} . \tag{8.5}$$

With a machine precision of 10^{-16} and these bounds, the norm of the uncertainty of **x** potentially exceeds the norm of **x**. Under these conditions, it appears that the model may be unable to produce meaningful results. The penalty method of solving the FEM equations is introducing entries of the order 10^{30} in **A**. Other terms in **A** tend to have magnitudes of 10^{9} to 10^{11}. Two possibilities could be investigated. The FEM problem can be solved without introducing the 10^{30} entries. It would be interesting to know the condition number of the matrix without them. Assuming this condition number is small enough to ensure meaningful results from solving the system of equations, it would then be interesting to compare those results with the results produced by SuperLU and the penalty method.

With scaling, $\kappa = 2.63 \cdot 10^{5}$. While the matrix is better scaled for computing **x**, the ill-conditioning of the actual problem still exists. SuperLU also computes *BERR* and *FERR*. With scaling and iterative refinement, these values across the four column permutation methods are

$$3.58 \cdot 10^{-16} \leq BERR \leq 4.83 \cdot 10^{-16} \tag{8.6}$$

and

$$FERR = 2.02 \cdot 10^{-10} . \tag{8.7}$$

BERR is in line with the *BERR* numbers computed by `ccberr.f` and reported in the above table. *FERR* is in line with the product of κ and the machine precision. SuperLU only reports *BERR* or *FERR* when iterative refinement is performed. It appears that MMD on $\mathbf{A}^{\mathrm{T}} + \mathbf{A}$ with scaling and iterative refinement is the best choice for solving the 3-D model without pressure, but the matrix condition number remains an open issue.

Initial tests of SuperLU with the m3dp.15x15x5 matrix with pressure produced the following results.

Column Permutation	Scale	I.R.	BERR	||R||infinity	FLOPs	Time (secs)	Memory (MB)
Natural	No	No	1.78E-12	8.78E-03	3.56E+09	4.70	50.1
Natural	Yes	No	1.56E-12	8.95E-03	3.56E+09	4.74	50.1
Natural	Yes	Yes	8.03E-16	1.87E-03	3.56E+09	5.10	50.1
MMD A^T+A	No	No	7.14E-11	1.58E-01	2.26E+10	30.9	113.4
MMD A^T+A	Yes	No	1.83E-13	3.09E-02	1.76E+10	23.1	103.1
MMD A^T+A	Yes	Yes	7.89E-16	2.19E-03	1.76E+10	25.5	103.1
MMD A^TA	No	No	7.45E-12	1.56E-02	1.04E+09	1.74	28.6
MMD A^TA	Yes	No	1.00	4.09E+15	9.55E+08	1.64	27.4
MMD A^TA	Yes	Yes	4.25E-12	2.07E+01	9.55E+08	1.81	27.4
COLAMD	No	No	7.44E-12	2.72E-02	1.39E+09	3.04	36.9
COLAMD	Yes	No	1.08E-12	3.93E-02	1.36E+09	3.00	36.3
COLAMD	Yes	Yes	8.08E-16	2.17E-03	1.36E+09	3.24	36.3

Table 8. 7. Initial SuperLU tests with m3dp.20x20x5.

MMD on $\mathbf{A}^{\mathbf{T}} + \mathbf{A}$, which was best for the problem without pressure, now produces the worst results in terms of FLOPs, time and memory for this matrix. SuperLU reports multiple memory expansions occurred. MMD on $\mathbf{A}^{\mathbf{T}}\mathbf{A}$ produces the best results in terms of FLOPS, time, and memory, but the error measures are high. This leaves COLAMD as the seemingly best choice. The effect of scaling is not as pronounced for this matrix, but the effect of iterative refinement is more pronounced.

The condition number of the matrix without scaling is $\kappa = 4.33 \cdot 10^{23}$. With scaling, $\kappa = 5.12 \cdot 10^{3}$. The same concern we had for the large condition number in the problem without pressure also applies to the problem with pressure.

8.6. Initial Tests of UMFPACK

Initial tests of UMFPACK with the m3d.20x20x5 matrix produced the following results.

Column Permutation	Scale	I.R.	BERR	||R||infinity	κ	FLOPS	Time (secs)	Memory (MB)
AMD A^T+A	No	No	1.40E-15	9.26E-03	5.78E+19	1.51E+09	1.12	37.3
AMD A^T+A	No	Yes	3.75E-16	4.47E-03	5.78E+19	1.53E+09	1.18	37.3
AMD A^T+A	Yes	No	1.11E-15	8.29E-03	7.04E+04	1.51E+09	1.12	37.3
AMD A^T+A	Yes	Yes	3.24E-16	6.15E-03	7.04E+04	1.53E+09	1.19	37.3
COLAMD	No	No	3.73E-14	1.69E-02	1.50E+20	2.16E+09	1.62	44.7
COLAMD	No	Yes	3.17E-16	3.85E-03	1.50E+20	2.18E+09	1.70	44.7
COLAMD	Yes	No	1.75E-03	4.58E+09	5.38E+04	1.11E+09	0.99	29.6
COLAMD	Yes	Yes	3.80E-16	4.81E-03	5.38E+04	1.13E+09	1.05	29.6

Table 8. 8. Initial UMFPACK tests with m3d.20x20x5.

UMFPACK chooses AMD on $\mathbf{A}^T + \mathbf{A}$ as the column permutation strategy when left to pick the strategy on its own. While both permutation strategies deliver similar results with scaling and iterative refinement, error measures for COLAMD with scaling and no iterative refinement are anomalous. Scaling does a good job reducing fill-in with COLAMD resulting in reduced FLOPS, time, and memory, but does nothing for the AMD on $\mathbf{A}^T + \mathbf{A}$ strategy.

The condition number estimates computed by the two strategies are comparable. They are also pretty much in line with SuperLU's estimates.

Initial tests of SuperLU with the m3dp.15x15x5 matrix produced the following results.

Column Permutation	Scale	I.R.	BERR	||R||infinity	κ	FLOPs	Time (secs)	Memory (MB)
AMD A^T+A	No	No	6.41E-11	4.64E-01	2.89E+23	3.41E+09	5.58	52.6
AMD A^T+A	No	Yes	5.73E-16	1.45E-03	2.89E+23	3.43E+09	5.65	52.6
AMD A^T+A	Yes	No	2.32E-11	6.79E-01	7.46E+05	2.84E+09	4.22	49.9
AMD A^T+A	Yes	Yes	5.55E-16	1.32E-03	7.46E+05	2.87E+09	4.35	49.9
COLAMD	No	No	3.39E-11	1.58E+00	9.43E+23	1.15E+09	1.00	28.1
COLAMD	No	Yes	4.91E-16	1.41E-03	9.43E+23	1.17E+09	1.05	28.1
COLAMD	Yes	No	1.00	2.30E+15	2.54E+03	6.10E+08	0.65	19.6
COLAMD	Yes	Yes	2.03E-11	5.22E+01	2.54E+03	6.22E+08	0.70	19.6

Table 8. 9. Initial UMFPACK tests with m3dp.15x15x5.

UMFPACK chooses COLAMD as the column permutation strategy when left to pick the strategy on its own. Unfortunately, the COLAMD strategy again has anomalous error measures with scaling. Otherwise, COLAMD has good performance specs. Runtime for AMD on $\mathbf{A}^T + \mathbf{A}$ is particularly long and not it proportion to the number of FLOPs performed or the memory used. The answer appears to be that UMFPACK's initial memory allocation is too low, causing an excessive number of memory reallocations.

More work should be done to determine why the COLAMD error measures are anomalous. This may be a software bug and not an inherent limitation in the algorithm. Comparing the solution vectors produced by COLAMD and other methods would be a good place to start. For COLAMD with scaling and iterative refinement, UMFPACK reports $BERR = 5.31 \cdot 10^{-16}$ instead of $BERR = 2.03 \cdot 10^{-11}$ reported by `ccberr.f` and shown in the table above. Usually these measures are well within a factor of ten of each other.

8.7. Detailed Test Results

The software for solving our systems of equations should be fast, should have modest memory requirements, and should produce stable calculations with small error measures. The number of non-zero entries in the **L** and **U** factors is a key parameter in evaluating speed and memory requirements of an algorithm. The number of floating point operations and the amount of memory required tend to increase with increasing numbers of non-zero entries, and runtime tends to increase with increasing numbers of floating point operations. In the next two sections, we will look at numbers of non-zero entries in L and U, total memory requirements, number of floating point operations, and runtime. These numbers will be compared for SuperLU, UMFPACK, BGAUSS and BLU. The first section evaluates these methods in terms of the ice sheet model without pressure calculations (banded matrices). The second section evaluates them in terms of the ice sheet model with pressure calculations (unbanded matrices). The speedup produced by using BLAS is also investigated by looking at the number of floating point operations performed per second. The test results were produced using the test matrices discussed earlier. SuperLU and UMFPACK were run with the scaling and iterative refinement options enabled.

Issues regarding stability and error measures have been discussed in the sections on initial testing are not discussed further.

8.7.1. Detailed Results: Systems without Pressure

Figure 8.1 shows the number of non-zero entries in **L+U** as a function of problem size. Banded **LU** factorization and SuperLU with natural column ordering produce equal numbers of non-zero entries and produce more non-zeros than any other method. UMFPACK produces the second highest number of non-zero's. UMFPACK was allowed to automatically select its strategy for these tests. It chose to use its symmetric AMD on $\mathbf{A}^T + \mathbf{A}$ algorithm. SuperLU's COLAMD (labeld SLU-AMD) and MMD on $\mathbf{A}^T\mathbf{A}$ methods produce numbers of non-zero entries similar to UMFPACK for problem sizes of 13,500 rows and fewer, but they produce nearly 20% fewer non-zero entries for 24,000 rows. SuperLU's MMD on $\mathbf{A}^T + \mathbf{A}$ algorithm produces the fewest number of non-zeros by far, beating BLU and SuperLU natural ordering by nearly 50%.

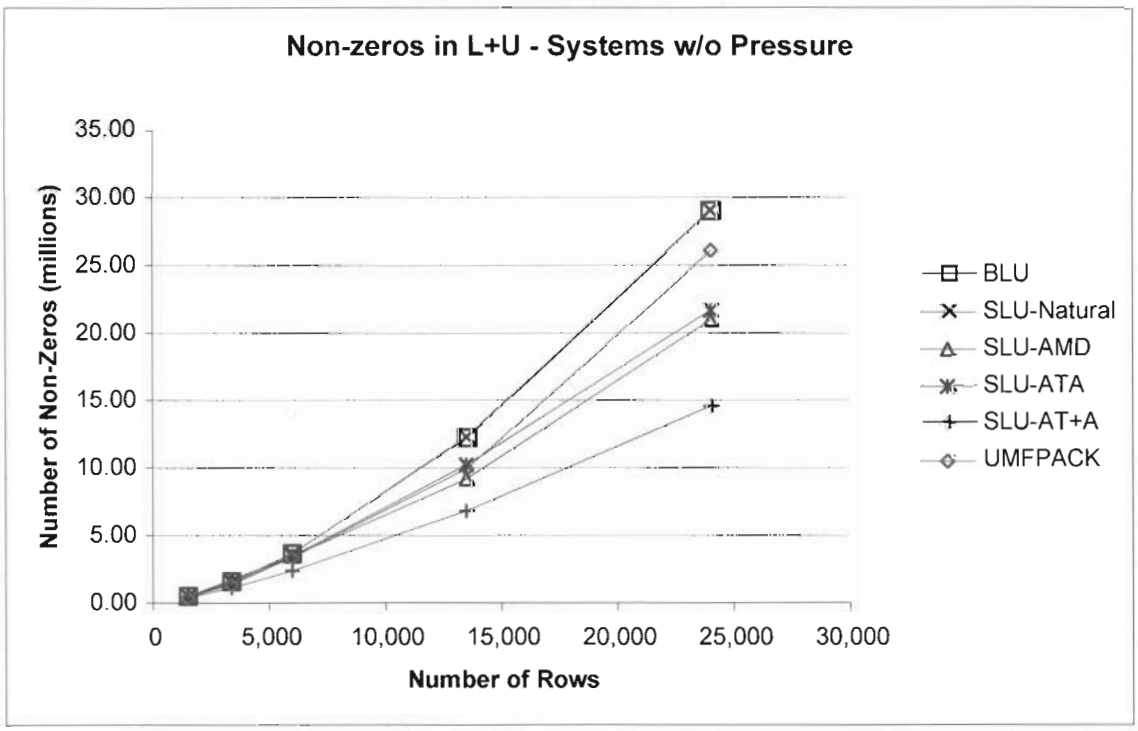

Figure 8. 1. Non-zeros in **L+U** in systems without pressure.

Figure 8.2 shows the total memory usage. As expected, UMFPACK and SuperLU memory usage is roughly proportional to the number of non-zeros in **L**+**U**. BLU and BGAUSS are different. The memory used by these methods is simply the bandwidth times the number of rows times eight bytes per double precision real number. Approximately 97% of the in-band zero entries become non-zeros in the BLU factorization. UMFPACK and SuperLU require working memory in addition to the memory required to store the **L** and **U** factors. Despite this, SuperLU's COLAMD, MMD on $\mathbf{A}^T\mathbf{A}$, and MMD on $\mathbf{A}^T + \mathbf{A}$ column ordering methods result in lower memory usage than BGAUSS and BLU.

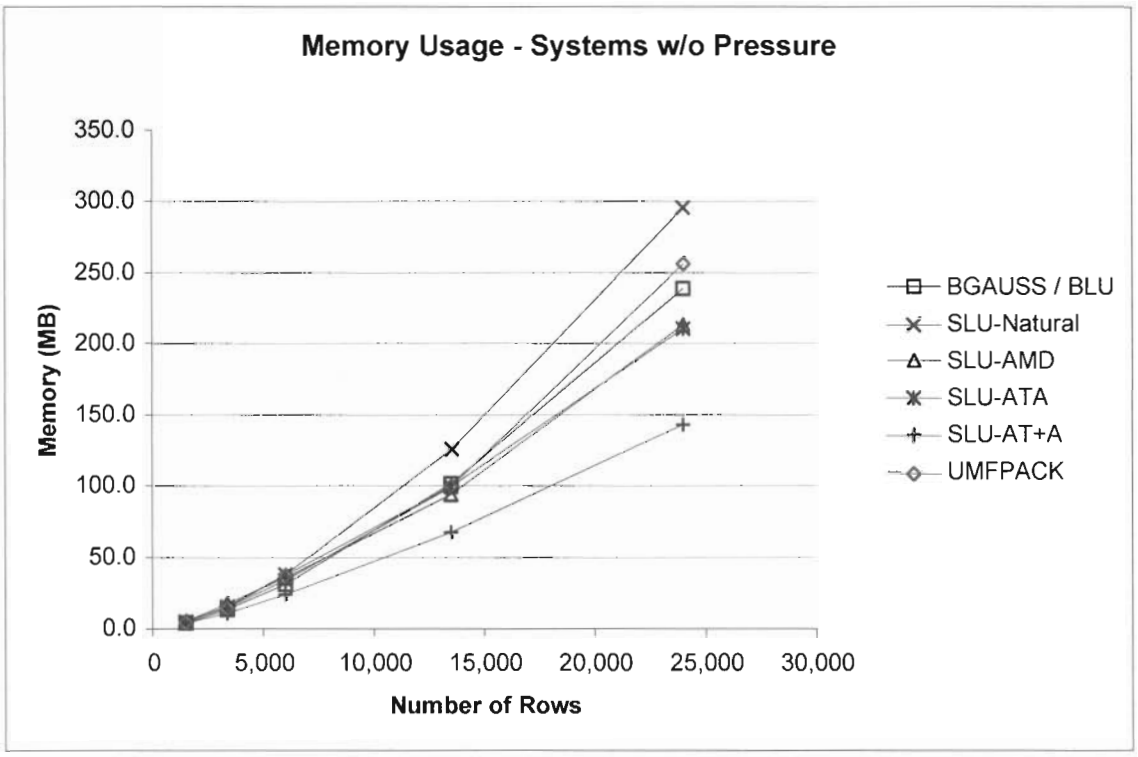

Figure 8. 2. Memory usage in systems without pressure.

Figure 8.3 shows the number of floating point operations performed. BGAUSS, BLU, and SuperLU with natural column ordering have approximately the same number of floating point operations for each problem size. UMFPACK is similar for problem sizes less than 24,000 rows, but is significantly higher for 24,000 rows. Although SuperLU COLAMD and SuperLU AMD on $\mathbf{A}^T\mathbf{A}$ have similar numbers of non-zeros in their **L** and **U** factors, MMD on $\mathbf{A}^T\mathbf{A}$ performs significantly more floating point operations. SuperLU's MMD on $\mathbf{A}^T + \mathbf{A}$ consistently performs fewer floating point operation than all other methods.

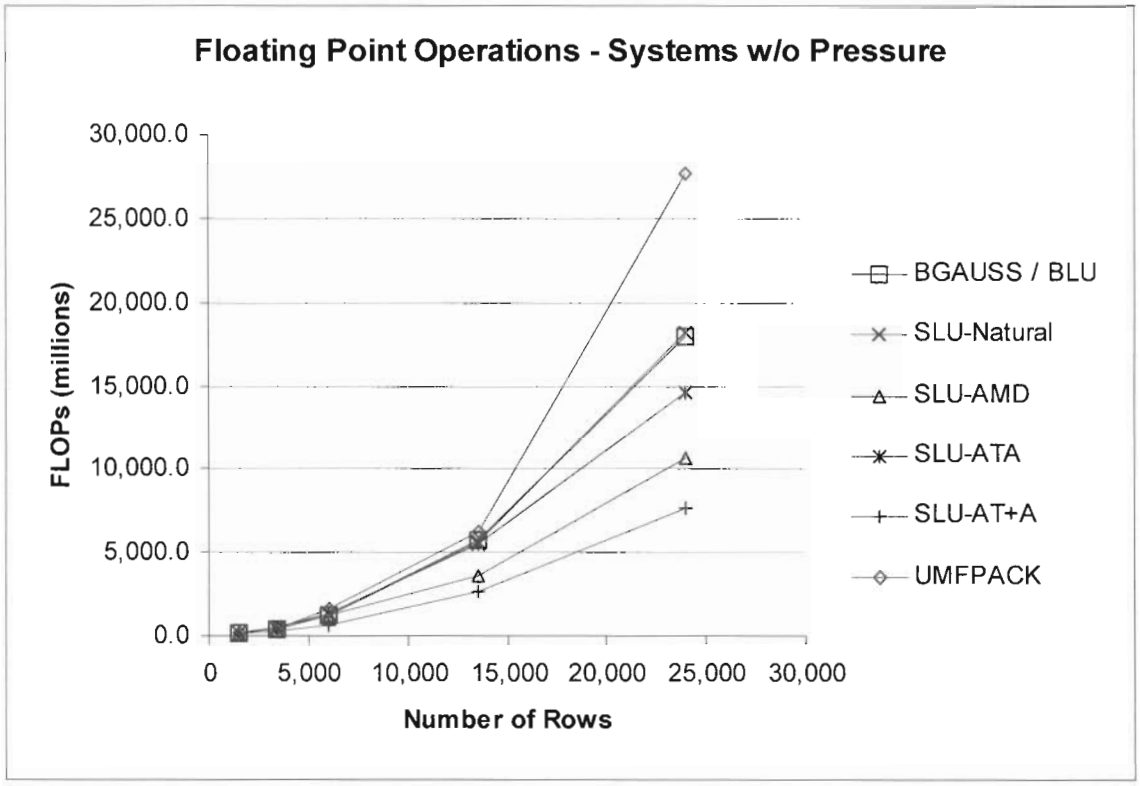

Figure 8. 3. Floating point operations in systems without pressure.

Figures 8.4 and 8.5 show CPU time as a function of problem size. Figure 8.4 includes BGAUSS, BLU and UMFPACK without BLAS data. Figure 8.5 leaves out these data so that the remaining methods can be seen in better detail. The most striking feature of Figure 8.4 is the reduction in CPU time for UMFPACK when BLAS are used. For 24,000 rows UMFPACK runs in approximately 1/8 the time when BLAS are used. Measurements for SuperLU without BLAS were not made. However, the speedup can be estimated by comparing the runtime of BLU to the runtime of SuperLU with natural column ordering. Each of these methods perform an equal number of floating point operations. BLU runs without BLAS and SuperLU runs with BLAS. The speedup from BLU to SuperLU natural ordering is approximately 2.5 for 24,000 rows. This is reasonable in light of UMFPACK's speedup and the fact that UMFPACK relies heavily on level 3 BLAS functions whereas SuperLU relies on level 2 BLAS functions. A high performance BLAS is important for optimum performance of UMFPACK and SuperLU.

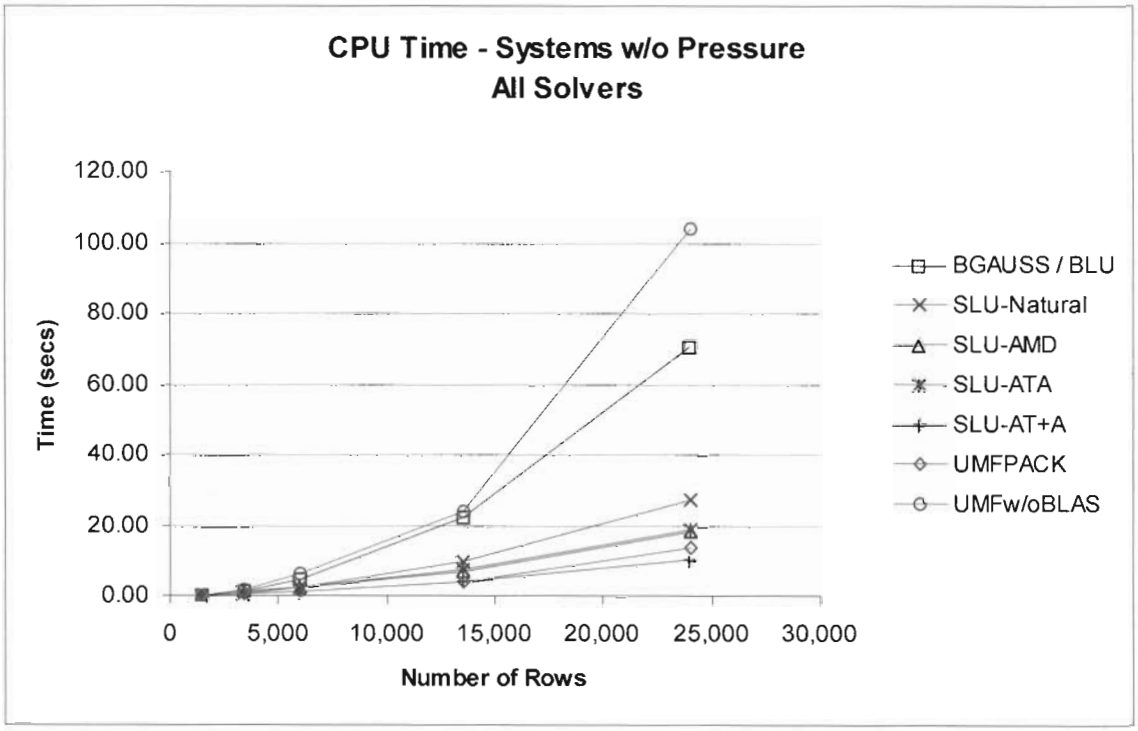

Figure 8. 4. CPU time for all solvers in systems without pressure.

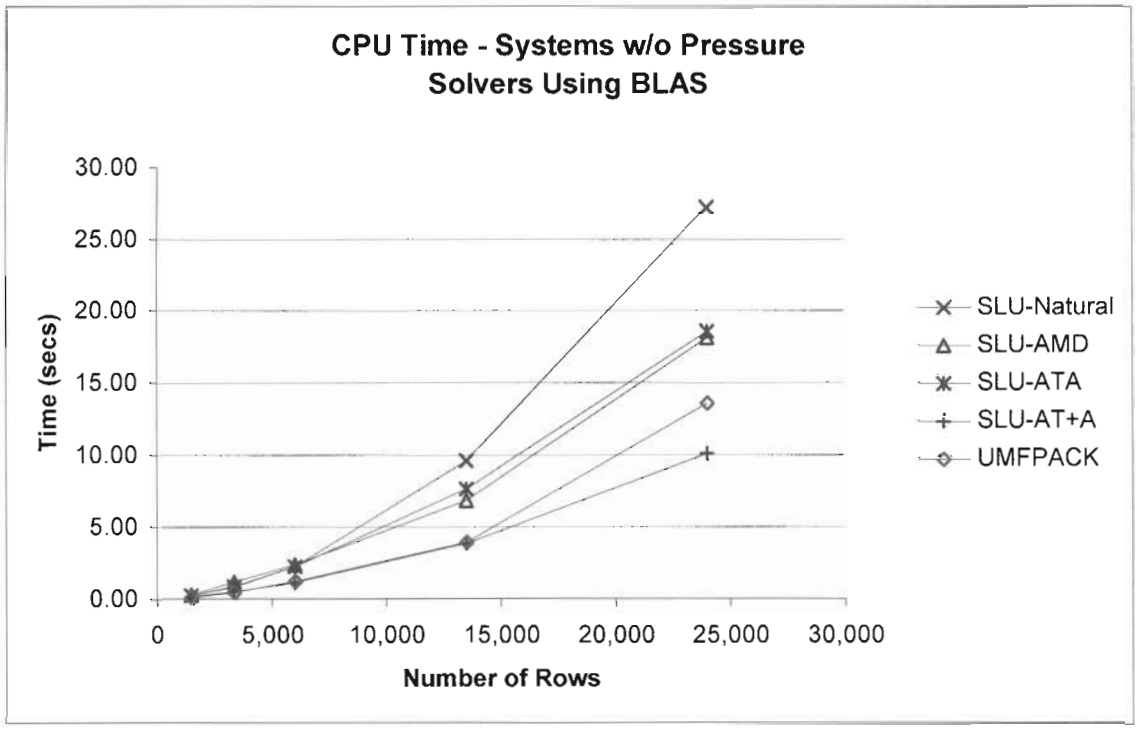

Figure 8. 5. CPU time for solvers with BLAS in systems without pressure.

In Figure 8.5 we see that SuperLU's performance is not just a function of the number of floating point operations. SuperLU COLAMD and SuperLU MMD on $\mathbf{A}^T\mathbf{A}$ have similar runtimes, but significantly different numbers of floating point operations. Neither method required memory expansions, which can be a source of additional runtime.

Figure 8.6 shows millions of floating point operations per second. Unsurprisingly, UMFPACK is at the top of the chart. In fact, its mega-FLOPS performance for 24,000 rows seems exceedingly high for an Intel 2.66 GHz Pentium 4 processor, but all the numbers seem to indicate that it is correct. Without a high performance BLAS, UMFPACK megaflop rates are comparable to BGAUSS and BLU. SuperLU performance tends to be restricted to a relatively narrow band. SuperLU COLAMD seems to have greater overhead than SuperLU with MMD on $\mathbf{A}^T\mathbf{A}$ and SuperLU with MMD on $\mathbf{A}^T + \mathbf{A}$.

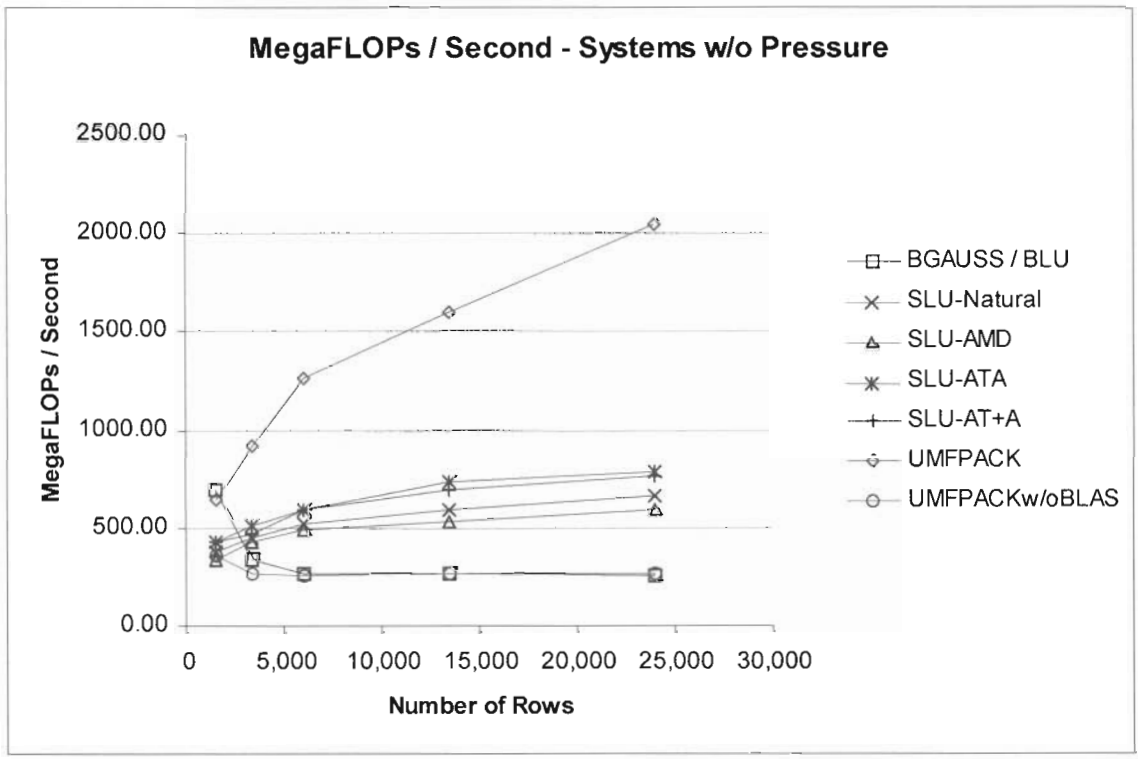

Figure 8. 6. MegaFLOPs per second in systems without pressure.

8.7.2. Detailed Results: Systems with Pressure

Figure 8.7 shows the number of non-zero entries in **L+U** versus problem size and method for problems with pressure. BGAUSS and BLU are not included because the ice sheet model with pressure does not produce banded matrices. Runtimes for SuperLU with natural column order and SuperLU with MMD on $\mathbf{A}^T + \mathbf{A}$ produced exceedingly long runtimes, so results are only presented for small problem sizes.

UMFPACK was again allowed to choose its own column ordering strategy and it chose the unsymmetric COLAMD strategy for this set of problems. UMPACK produces fewer non-zero entries than any of the SuperLU column ordering methods for this problem. SuperLU's MMD on $\mathbf{A}^T\mathbf{A}$ and COLAMD produce similar numbers of non-zero entries, just as they did in the problem without pressure. SuperLU's MMD on $\mathbf{A}^T + \mathbf{A}$, however, is now producing the greatest number of non-zero entries. It produced the least number of non-zeros for problems without pressure.

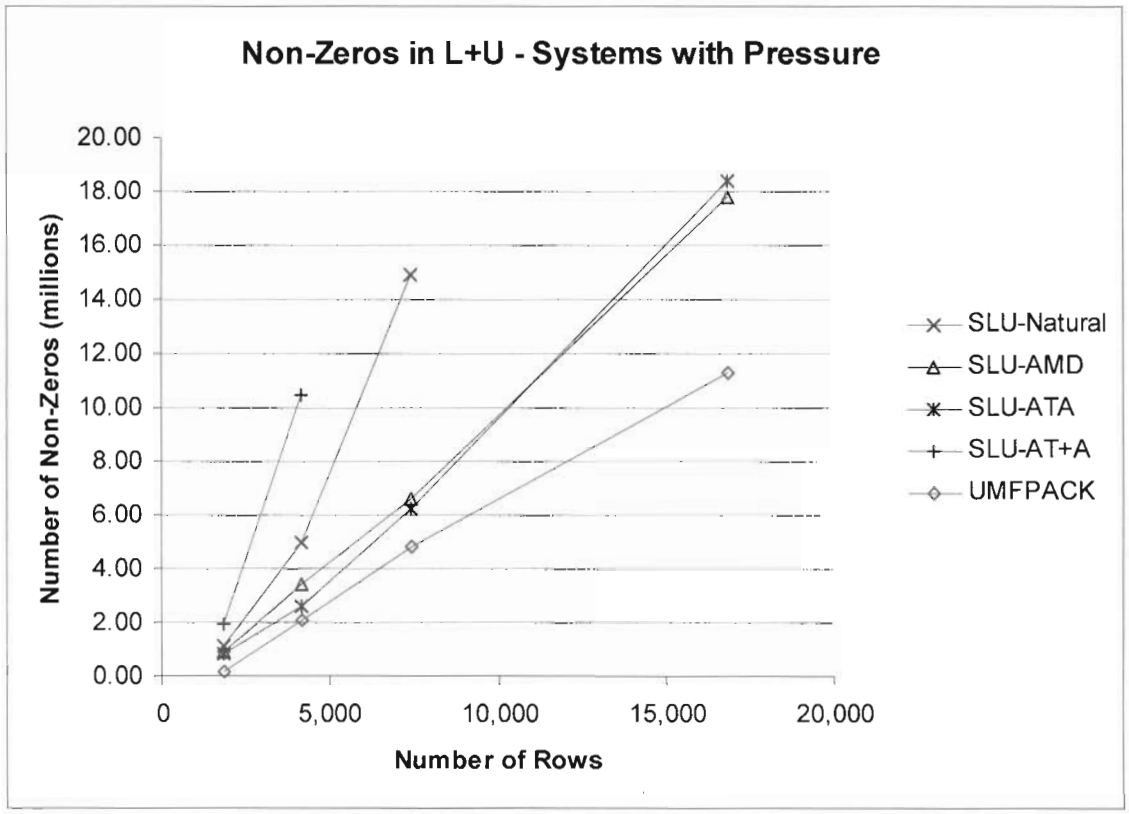

Figure 8. 7. Non-zeros in **L+U** in system with pressure.

Figure 8.8 shows memory usage as a function of problem size. These results are pretty much in line with the number of non-zeros in **L+U**. One exception is the difference in memory required by SuperLU with MMD on $\mathbf{A}^T\mathbf{A}$ and SuperLU with COLAMD. The COLAMD memory requirement is much less than the MMD on $\mathbf{A}^T\mathbf{A}$ memory requirement, even though both methods produce similar numbers of non-zero entries in **L+U**.

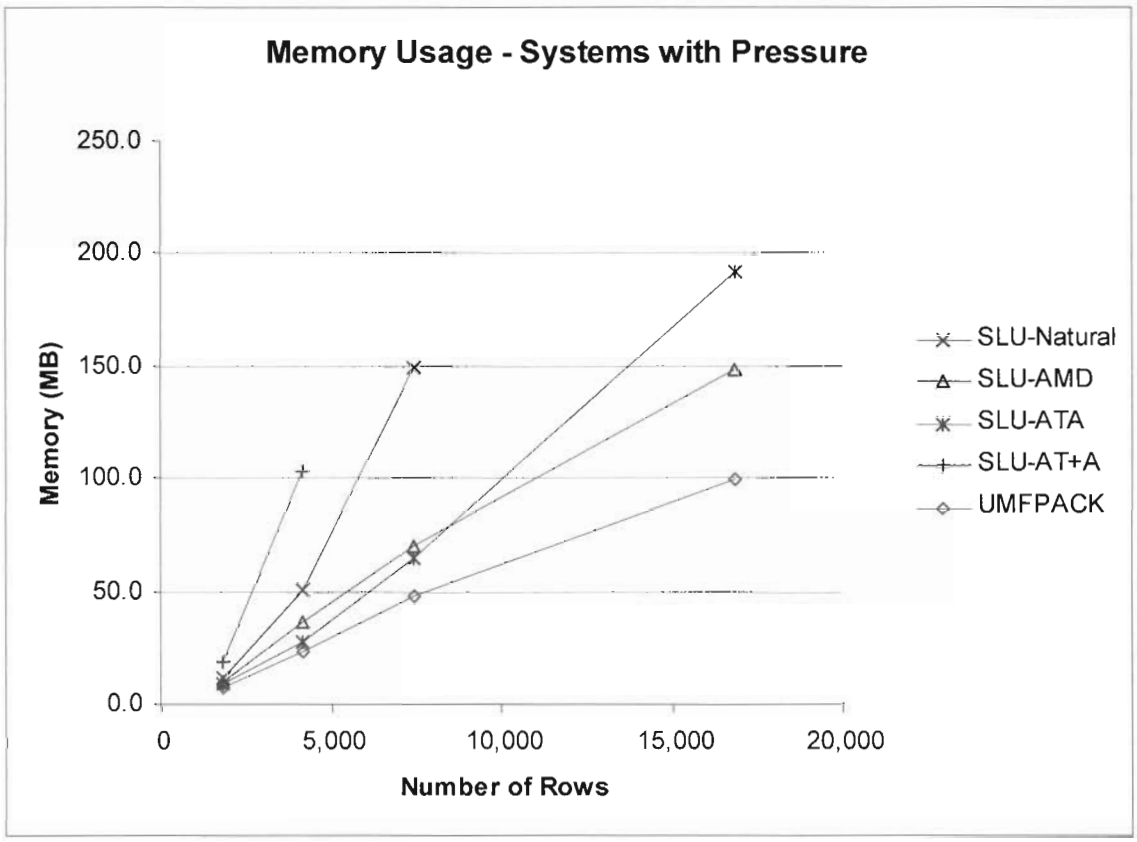

Figure 8. 8. Memory usage in systems with pressure.

Figure 8.9 shows the number of floating point operations as a function of problem size. UMFPACK's relative performance gets better for the largest problem. The number of floating point operations is more directly related to memory usage than it is to number of non-zero entries in **L+U**, but all three are similar.

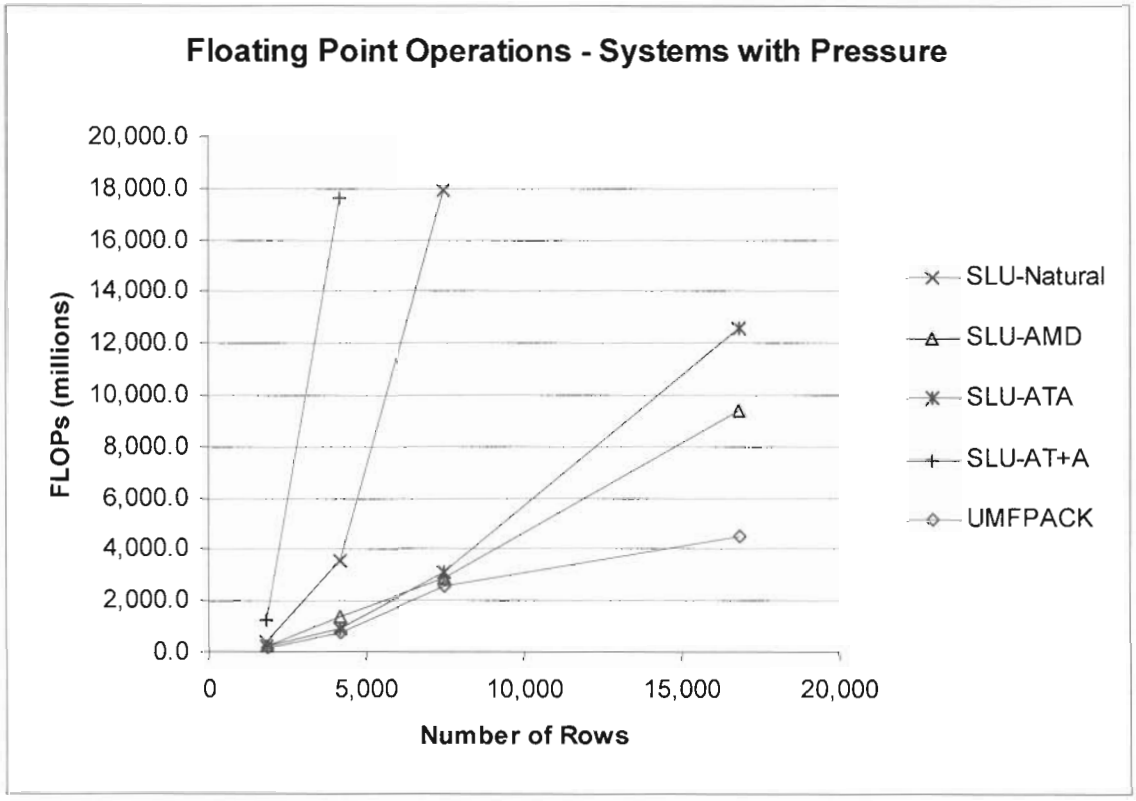

Figure 8. 9. Floating point operations in systems with pressure.

Figure 8.10 shows CPU time as a function of problem size. Again, there are no big surprises here. UMFPACK is fastest.

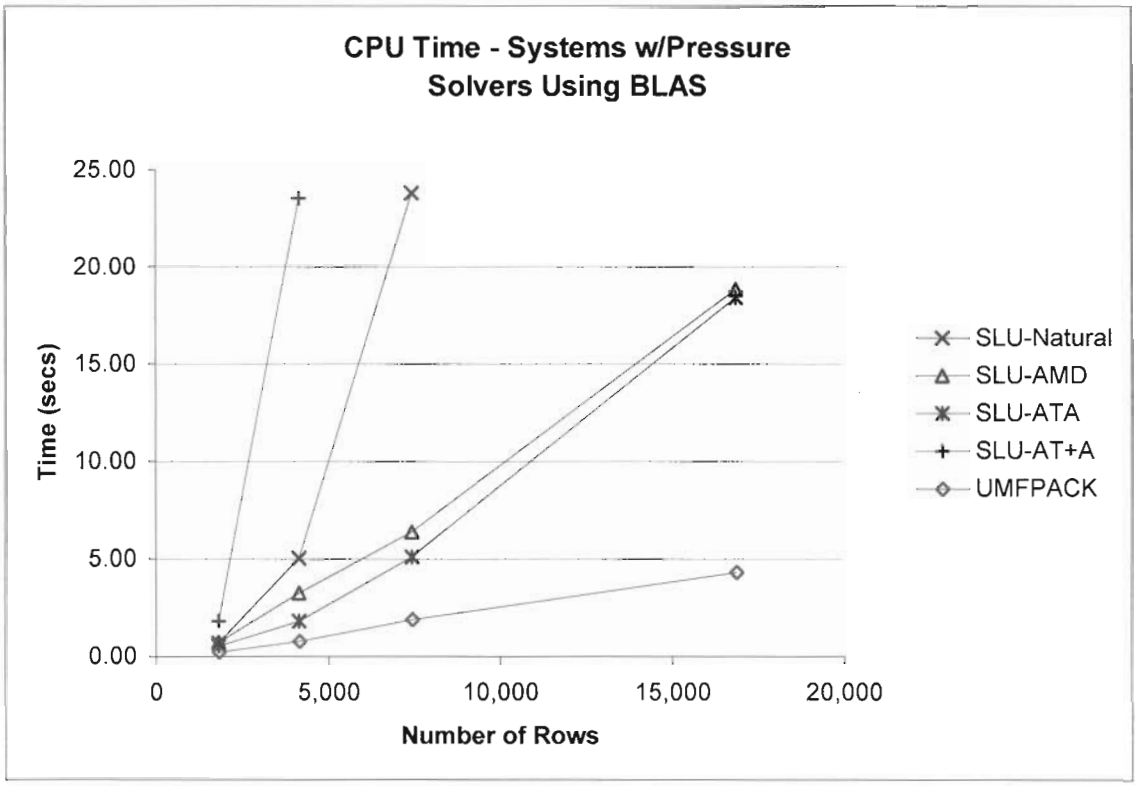

Figure 8. 10. CPU time in systems with pressure.

Finally, Figure 8.11 shows millions of floating point operations per second as a function of problem size. UMFPACK performance decreases for the largest problem size. This is an unexpected result. The decrease was not due to memory expansion. The initial memory allocation was sufficient for solving the problem. UMFPACK's solution for this problem has $BERR = 2.04 \cdot 10^{-10}$ while smaller problems had $BERR \approx 5 \cdot 10^{-16}$. It appears that something else is going on here. SuperLU with MMD on $\mathbf{A}^{\mathbf{T}} + \mathbf{A}$ also shows decreasing performance for the largest problem solved. In this case, however, there were nine memory expansions that occurred. Zero memory expansions tend to be the norm, and the problem size before the largest one had 3 memory expansions, so the need for memory expansions is assumed to be the cause of this performance decrease.

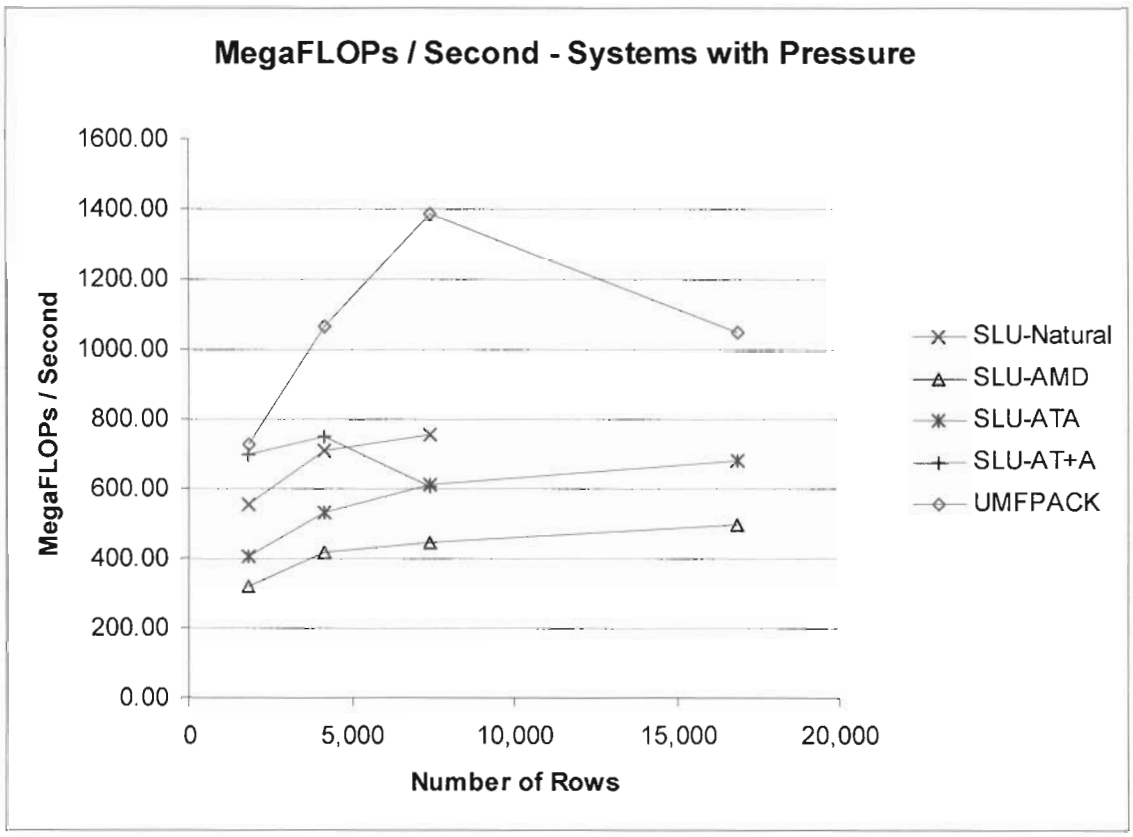

Figure 8. 11. MegaFLOPS in systems with pressure.

9. Conclusions and Future Work

SuperLU and UMFPACK are reasonable software packages for solving the systems of linear equations generated by the 3-D version of the University of Maine Ice Sheet Model. These packages, along with the interfacing routines and data structures developed in this work, meet the goals of running quickly and using main memory sparingly. In addition, they are capable of providing further insight into a system of equations by producing error measures and an estimate of the matrix condition number. SuperLU also offers the opportunity for the ice sheet model to be easily parallelized should the need arise.

An unsettling finding was the discovery that some column permutation methods produced solutions with unexpectedly high error measures. The reason is unknown. The matrices produced by the ice sheet model have large condition numbers that are on the order of 10^{21} to 10^{23}. Such large condition numbers cast concern on the meaning, or lack of meaning, of the computed solutions. They may also be the reason some column permutation methods are failing. The large condition numbers may be the result of using the penalty method to specify boundary conditions with the finite element method. This method introduces entries in the matrix that are on the order of 10^{30} while other entries tend to be on the order of 10^{9} to 10^{11}. The boundary conditions can be specified without introducing these large entries. Future work should investigate if the condition numbers can be reduced, and if reducing them takes care of the high error measures observed with some column permutation methods. If the condition number cannot be reduced, then we need to determine if solutions are indeed meaningful. While these findings are unsettling, it is at least reassuring that these software packages have brought them to light.

If we turn from these concerns and focus on the methods that produce low error measures, then we can draw the following conclusions. For problems without pressure, SuperLU's MMD on $\mathbf{A}^{\mathbf{T}} + \mathbf{A}$ column ordering method produces superb results. For the largest problem size, it runs seven times faster than banded Gaussian elimination and banded **LU** factorization. In addition to running faster, it also uses 40% less memory than banded Gaussian elimination and banded **LU** factorization.

For problems with pressure, UMFPACK's COLAMD column ordering strategy produces the fastest runtimes and uses the least amount of memory. Unfortunately, UMFPACK's error measures are very high. This leaves SuperLU with its COLAMD column ordering strategy as the best choice. Unfortunately, SuperLU's runtime is 4.4 times longer than UMFPACK's, and it uses 49% more memory. For future work, UMFPACK's AMD on $\mathbf{A}^T + \mathbf{A}$ column order strategy should be tested on matrices with pressure to see if it has better performance than SuperLU while maintaining low error measures.

Overall, SuperLU is an outstanding package. It offers a choice of four column ordering strategies. Its matrix equilibration algorithm normalizes both rows and columns of the matrix, which had a significant impact on ice sheet problems without pressure. In addition to providing an estimate of the matrix condition number and componentwise backward error, SuperLU also provides an estimate of forward error and a measure of pivot growth. On top of this, versions of SuperLU are available for shared memory and distributed memory parallel computers as well as single processor computers.

Despite its positives, SuperLU did not give a good initial impression. The user documentation is weak. At times the comments in the source code must be consulted to understand how to use the software. The simple driver routine lures the new user for an initial implementation, while the expert driver routine is really needed to use SuperLU's features. Matrix equilibration and iterative refinement are not enabled by default, but these features are necessary to get the best results. There are no guides for selecting a column ordering strategy. The user must experiment to determine what works best.

What SuperLU lacks in its initial impression, UMFPACK has. The UMFPACK user documentation is very good. Its default options include matrix scaling, iterative refinement, and automatic selection of a column ordering strategy. UMFPACK worked well the first time it was run. However, additional experience with the package reveals potential weaknesses for some problems. Its matrix scaling algorithm normalizes rows only and was not as effective as SuperLU. For problems without pressure, it used more memory than banded Gaussian elimination instead of less. It also had relatively more problems with high error measures

than SuperLU. UMFPACK's megaflops per second performance was outstanding. However, the performance measures that matter most from a user's point of view are runtime and memory usage.

A high performance BLAS is necessary to get optimum performance from both SuperLU and UMFPACK.

This work stopped short of doing a detailed study of column ordering strategies. While a detailed knowledge of these strategies is not necessary to use the software packages, a better understanding of them might lead one to make better choices with less trial and error. This topic can also be added to the list of future work.

While potential problems have been uncovered and additional work remains to be done, this work has accomplished its initial goal of efficiently solving the linear systems of equations generated by the University of Maine 3-D ice sheet model. Software packages have been identified that are far more robust than anything that would have been written from scratch. Data structures and methods have been written that efficiently interface the ice sheet model with the software packages. Finally, detailed testing of the software packages has been performed using actual systems of equations generated by the ice sheet model.

BIBLIOGRAPHY

[1] ... (1995). *Numerical Linear Algebra.* Computational Science Education Project. Sponsored by U.S. Department of Energy. http://www.phy.ornl.gov/csep/

[2] ... (1997). *Basic Linear Algebra Subprograms: A Quick Reference Guide.* University of Tennessee, Oak Ridge National Laboratory, and Numerical Algorithms Group Ltd. Available at http://netlib.org/blas/.

[3] Amestoy, P. R., Davis, T. A., and Duff, I. S. (1996). *An Approximate Minimum Degree Ordering Algorithm.* SIAM J. Matrix Anal. Applic., 17(4), 886-905.

[4] Amestoy, P. R., Davis, T. A., and Duff, I. S. (2004). *Algorithm 837: AMD, An Approximate Minimum Degree Ordering Algorithm.* ACM Trans. Math Software, 30(3), 381-388.

[5] Davis, T. A. (2005). *UMFPACK Version 4.3.1 User Guide.* Tech Report TR-04-003, Dept. of Computer and Information Science and Engineering, Univ. of Florida. http://www.cise.ufl.edu/research/sparse/umfpack/.

[6] Davis, T. A., and Duff, I. S. (1994). *An Unsymetric-Pattern Multifrontal Method for Sparse LU Factorization.* Tech report TR-94-038, Dept. of Computer and Information Science and Engineering, Univ. of Florida.

[7] Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. (2000). *A Column Approximate Minimum Degree Ordering Algorithm.* Tech Report TR-00-005, Dept. of Computer and Information Science and Engineering, Univ. of Florida. Submitted to ACM Trans. Math. Software.

[8] Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, Xiaoye S., and Liu, J. W. H. (1999). *A Supernodal Approach to Sparse Partial Pivoting.* Siam J. Matrix Analysis and Applications, 20, 720-755.

[9] Demmel, J. W., Gilbert, J. R., and Li, Xiaoye S. (1999, Last update 2003). *SuperLU Users' Guide.* Tech report LBNL-44289, Computational Research Division, Lawrence Berkley National Laboratory. http://crd.lbl.gov/~xiaoye/SuperLU/.

[10] Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). *Direct Methods for Sparse Matrices.* Oxford University Press.

[11] Golub, G. H., and Van Loan, C. F. (1996). *Matrix Computations, Third Edition.* The John Hopkins University Press.

[12] Lay, D. C. (2003). *Linear Algebra and Its Applications, Third Edition.* Addison Wesley.

[13] Li, Xiaoye S. (2005). *An overview of SuperLU: Algorithms, Implementation, and User Interface.* ACM Transactions on Mathematical Software, Vol. 31, No. 3, 302-325.

[14] Liu, J. W. (1985). *Modification of the Minimum Degree Algorithm by Multiple Elimination.* ACM Trans. Math. Software, 11, 141-153.

APPENDICIES

Appendix 1. Modified Compressed Column Routines

A1.1. ccadd.f

```
C-----------------------------------------------------------------------
C   ccadd.f
C
C   03/13/2005  Rodney Jacobs
C
C   Add a value to an element of a matrix stored in compressed column
C   format.
C
C   Input:
C     IROW       row of element to return
C     ICOL       column of element to return
C     VAL        value of element to store
C     ICCPTR     compressed column pointers
C     ICCROW     compressed column row indices
C     CCVAL      compressed column matrix element values
C     NCOL       number of columns in matrix
C
C   Return:
C     New value of the array element
C
C   Side effects:
C     ICCPTR, ICCROW, CCVAL, and NZ are updated
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      FUNCTION CCADD (IROW, ICOL, VAL, ICCPTR, ICCROW, CCVAL, NCOL)
      IMPLICIT REAL*8 (A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCVAL(MAXNZ)
C ---
      J0=ICCPTR(ICOL,1)
      J1=ICCPTR(ICOL,2)
      J2=J1
C ---
C --- Binary search for IROW at ICOL
C ---
      DO WHILE (J0.LE.J1)
        JMID=(J0+J1)/2
        IR=ICCROW(JMID)
        IF (IR.EQ.IROW) THEN
          CCVAL(JMID)=CCVAL(JMID)+VAL
          CCADD=CCVAL(JMID)
          RETURN
        ENDIF
        IF (IR.LT.IROW) THEN
          J0=JMID+1
        ELSE
          J1=JMID-1
        ENDIF
      ENDDO
C ---
C --- New matrix element must be added
C ---
C --- Expand the bucket for <ICOL> by 20% if it is full
C ---
      IF (J2+1.GE.ICCPTR(ICOL+1,1)) THEN
        JDELTA=0.2*(ICCPTR(ICOL,2)-ICCPTR(ICOL,1)+1)
        IF (JDELTA.LT.1) JDELTA=1
```

```
        IF (ICCPTR(NCOL+1,2)+JDELTA.GT.MAXNZ) THEN
          PRINT *,
     1      'CCADD: Compressed column structure size exceeds MAXNZ'
          STOP
        ENDIF
        I=NCOL+1
        DO WHILE (I.GT.ICOL)
          K0=ICCPTR(I,1)
          K1=ICCPTR(I,2)
          K2=K1+JDELTA
          DO WHILE (K1.GE.K0)
            ICCROW(K2)=ICCROW(K1)
            CCVAL(K2)=CCVAL(K1)
            K2=K2-1
            K1=K1-1
          ENDDO
          ICCPTR(I,1)=ICCPTR(I,1)+JDELTA
          ICCPTR(I,2)=ICCPTR(I,2)+JDELTA
          I=I-1
        ENDDO
      ENDIF
C ---
C --- Add the new element
C ---
      DO WHILE (J2.GE.J0)
        ICCROW(J2+1)=ICCROW(J2)
        CCVAL(J2+1)=CCVAL(J2)
        J2=J2-1
      ENDDO
      ICCROW(J0)=IROW
      CCVAL(J0)=VAL
      ICCPTR(ICOL,2)=ICCPTR(ICOL,2)+1
      CCADD=VAL
      END
```

A1.2. ccbase0.c

```
/* ccbase0.c
 *     Convert compressed column format from base 1 for Fortran to base 0
 *       for C.
 *
 *     Input:
 *       nz:          number of matrix elements stored
 *       nrow:        number of rows
 *       ccptr[]:     compressed column pointers
 *       ccrow[]:     row indices
 *     Output:
 *       ccptr[]:     compressed column pointers
 *       ccrow[]:     row indices
 *
 *     Rodney Jacobs, University of Maine, 2005
 */

void ccbase0 (int nz, int nrow, int ccptr[], int ccrow[]) {

    int i;

    for (i=0; i<=nrow; i++) --ccptr[i];
    for (i=0; i<nz; i++) --ccrow[i];
}
```

A1.3. ccbase1.c

```
/* ccbase1.c
 *     Convert compressed column format from base 0 for C to base 1
 *     for Fortran.
 *
 *     Input:
 *       nz:          number of matrix elements stored
 *       nrow:        number of rows
 *       ccptr[]:     compressed column pointers
 *       ccrow[]:     row indices
 *     Output:
 *       ccptr[]:     compressed column pointers
 *       ccrow[]:     row indices
 *
 *     Rodney Jacobs, University of Maine, 2005
 */

void ccbase1 (int nz, int nrow, int ccptr[], int ccrow[]) {

    int i;

    for (i=0; i<=nrow; i++) ++ccptr[i];
    for (i=0; i<nz; i++) ++ccrow[i];
}
```

A1.4. ccget.f

```
C-----------------------------------------------------------------------
C   ccget.f
C
C   03/13/2005  Rodney Jacobs
C
C   Return the value of a matrix element from a compressed column
C   representation of the matrix.
C
C   Input:
C     IROW       row of element to return
C     ICOL       column of element to return
C     ICCPTR     compressed column pointers
C     ICCROW     compressed column row indices
C     CCVAL      compressed column matrix element values
C
C   Return:
C     Matrix element value at <IROW> and <ICOL>.  Zero is returned if
C     no matrix element value is stored for <IROW> and <ICOL>.
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      FUNCTION CCGET (IROW, ICOL, ICCPTR, ICCROW, CCVAL)
      IMPLICIT REAL*8 (A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCVAL(MAXNZ)
C ---
      J0=ICCPTR(ICOL,1)
      J1=ICCPTR(ICOL,2)
C ---
C --- Binary search for IROW at ICOL
C ---
      DO WHILE (J0.LE.J1)
        JMID=(J0+J1)/2
        IR=ICCROW(JMID)
        IF (IR.EQ.IROW) THEN
          CCGET=CCVAL(JMID)
          RETURN
        ENDIF
        IF (IR.LT.IROW) THEN
          J0=JMID+1
        ELSE
          J1=JMID-1
        ENDIF
      ENDDO
C ---
C --- Element not found
C ---
      CCGET=0.D0
      END
```

A1.5. ccinit.f

```
C-----------------------------------------------------------------------
C   ccinit.f
C
C   03/13/2005  Rodney Jacobs
C
C   Initialize arrays for compressed column matrix storage
C
C   Input:
C     ICCPTR     compressed column pointers
C     NCOL       number of columns in matrix
C     NBKTSZ  numer of elements to allocate per row
C
C   Side effects:
C     ICCPTR(ICOL,1) is the starting index of ICCROW for elements in
C     column <ICOL> of the matrix.  ICCPTR(ICOL,2) is the ending index
C     of ICCROW for elements in column <ICOL>.  Initially, all columns
C     are empty, so ICCPTR(ICOL,2)=ICCPTR(ICOL,1)-1.  Initial values
C     of ICCPTR are set so that the estimated number of entries per
C     column can be added to the data structure without having to
C     move data in the structure.
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE CCINIT (ICCPTR, NCOL, NBKTSZ)
      IMPLICIT REAL*8 (A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2)
C
      IF (NCOL+1.GT.MAXCOL) THEN
        PRINT *,'CCINT: MAXCOL size exceeded',ncol,maxcol
        STOP
      ENDIF
      IF (NBKTSZ*NCOL.GT.MAXNZ) THEN
        PRINT *,'CCINT: MAXNZ size exceeded',nbktsz*ncol,maxnz
        STOP
      ENDIF
      DO I=1,NCOL+1
        ICCPTR(I,1)=1+(I-1)*NBKTSZ
        ICCPTR(I,2)=ICCPTR(I,1)-1
      ENDDO
      END
```

A1.6. ccparam.h

```
C-----------------------------------------------------------------------
C   ccparam.h
C
C   Parameters for compressed column matrix storage routines.
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
C ---
C --- MAXCOL is the maximum number of columns in the matrix + 1.
C --- MAXNZ is the maximum number of explicitly specified matrix element
C --- values.  Usually, these are non-zero values.
C ---
      PARAMETER(MAXCOL=40636,MAXNZ=4100000)
```

A1.7. ccput.f

```
C-----------------------------------------------------------------------
C   ccput.f
C
C   03/13/2005  Rodney Jacobs
C
C   Store the value of a matrix element in a compressed column
C   representation of the matrix.
C
C   Input:
C     IROW        row of element to return
C     ICOL        column of element to return
C     VAL         value of element to store
C     ICCPTR      compressed column pointers
C     ICCROW      compressed column row indices
C     CCVAL       compressed column matrix element values
C     NCOL        number of columns in matrix
C
C   Side effects:
C     ICCPTR, ICCROW, CCVAL, and NZ are updated
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE CCPUT (IROW, ICOL, VAL, ICCPTR, ICCROW, CCVAL, NCOL)
      IMPLICIT REAL*8 (A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCVAL(MAXNZ)
C ---
      J0=ICCPTR(ICOL,1)
      J1=ICCPTR(ICOL,2)
      J2=J1
C ---
C --- Binary search for IROW at ICOL
C ---
      DO WHILE (J0.LE.J1)
        JMID=(J0+J1)/2
        IR=ICCROW(JMID)
        IF (IR.EQ.IROW) THEN
          CCVAL(JMID)=VAL
          RETURN
        ENDIF
        IF (IR.LT.IROW) THEN
          J0=JMID+1
        ELSE
          J1=JMID-1
        ENDIF
      ENDDO
C ---
C --- New matrix element must be added
C ---
C --- Expand the bucket for <ICOL> by 20% if it is full
C ---
      IF (J2+1.GE.ICCPTR(ICOL+1,1)) THEN
        JDELTA=0.2*(ICCPTR(ICOL,2)-ICCPTR(ICOL,1)+1)
        IF (JDELTA.LT.1) JDELTA=1
        IF (ICCPTR(NCOL+1,2)+JDELTA.GT.MAXNZ) THEN
          PRINT *,
     1      'CCPUT: Compressed column structure size exceeds MAXNZ'
          STOP
        ENDIF
        I=NCOL+1
        DO WHILE (I.GT.ICOL)
```

```
          K0=ICCPTR(I,1)
          K1=ICCPTR(I,2)
          K2=K1+JDELTA
          DO WHILE (K1.GE.K0)
            ICCROW(K2)=ICCROW(K1)
            CCVAL(K2)=CCVAL(K1)
            K2=K2-1
            K1=K1-1
          ENDDO
          ICCPTR(I,1)=ICCPTR(I,1)+JDELTA
          ICCPTR(I,2)=ICCPTR(I,2)+JDELTA
          I=I-1
        ENDDO
      ENDIF
C ---
C --- Add the new element
C ---
      DO WHILE (J2.GE.J0)
        ICCROW(J2+1)=ICCROW(J2)
        CCVAL(J2+1)=CCVAL(J2)
        J2=J2-1
      ENDDO
      ICCROW(J0)=IROW
      CCVAL(J0)=VAL
      ICCPTR(ICOL,2)=ICCPTR(ICOL,2)+1
      END
```

A1.8. ccsqz.f

```
C-----------------------------------------------------------------------
C   ccsqz.f
C
C   03/13/2005  Rodney Jacobs
C
C   Remove free space from buckets of the compressed column data
C   structure.  This produces a conventional compressed column data
C   structure.
C
C   Input:
C     ICCPTR     compressed column pointers
C     ICCROW     compressed column row indices
C     CCVAL      compressed column values
C     NCOL       number of columns in matrix
C     NZ         number of matrix elements stored
C
C   Side effects:
C     NZ         number of matrix elements stored is updated
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE CCSQZ (ICCPTR, ICCROW, CCVAL, NCOL, NZ)
      IMPLICIT REAL*8 (A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCVAL(MAXNZ)
C ---
      DO I=1,NCOL
        I1=I+1
        J0=ICCPTR(I,2)+1
        J1=ICCPTR(I1,1)
        J2=ICCPTR(I1,2)
        JDELTA=J1-J0
        IF (JDELTA.GT.0) THEN
          DO WHILE (J1.LE.J2)
            ICCROW(J0)=ICCROW(J1)
            CCVAL(J0)=CCVAL(J1)
            J0=J0+1
            J1=J1+1
          ENDDO
          ICCPTR(I1,1)=ICCPTR(I1,1)-JDELTA
          ICCPTR(I1,2)=ICCPTR(I1,2)-JDELTA
        ENDIF
      ENDDO
      NZ=ICCPTR(NCOL+1,2)
      END
```

A1.9. cctest.f

```
C-----------------------------------------------------------------------
C   cctest.f
C
C   Test performance of compressed column routines.
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCVAL(MAXNZ)
      DIMENSION IROW(MAXNZ),ICOL(MAXNZ),VAL(MAXNZ)
      REAL DTIME,TA(2)
      SAVE ICCPTR,ICCROW,CCVAL,IROW,ICOL,VAL
C ---
C --- Pre-load data from data file into internal arrays
C ---
      PRINT *,'Test Performance of Compressed Column Routines'
      TIME=DTIME(TA)
      OPEN (1,FILE='matrix')
      READ (1,*) NE,NROW
      NCOL=NROW
      DO I=1,NE
        READ (1,*) IROW(I),ICOL(I),VAL(I)
      ENDDO
      CLOSE (1)
      TIME=DTIME(TA)
      NBKTSZ=MAXNZ/MAXCOL
      PRINT *,'MAXCOL...........: ',MAXCOL
      PRINT *,'MAXNZ............: ',MAXNZ
      PRINT *,'NBKTSZ...........: ',NBKTSZ
      PRINT *,'# of rows........: ',NROW
      PRINT *,'# of elements....: ',NE
      PRINT *,'Read time........: ',TIME
C ---
C --- Perform multiple iterations of the tests
C ---
      DO ITER=1,4
      PRINT *,' '
      PRINT *,'Iteration #',ITER
C ---
C --- Initialize compressed column data structure with CCINIT
C ---
      TIME=DTIME(TA)
      CALL CCINIT (ICCPTR,NCOL,NBKTSZ)
      TIME=DTIME(TA)
      PRINT *,'CCINIT time......: ',TIME
C ---
C --- Load compressed column data structure with CCPUT
C ---
      TIME=DTIME(TA)
      DO I=1,NE
        CALL CCPUT (IROW(I),ICOL(I),VAL(I),ICCPTR,ICCROW,CCVAL,NCOL)
      ENDDO
      TIME=DTIME(TA)
      PRINT *,'CCPUT time.......: ',TIME
C ---
C --- Squeeze data in compressed column arrays with CCSQZ
C ---
      TIME=DTIME(TA)
      CALL CCSQZ (ICCPTR,ICCROW,CCVAL,NCOL,NZ)
      TIME=DTIME(TA)
```

```
      PRINT *,'CCSQZ time.......: ',TIME
C ---
C --- Get data in compressed column data structure with CCGET
C ---
      TIME=DTIME(TA)
      DO I=1,NE
        A=CCGET (IROW(I),ICOL(I),ICCPTR,ICCROW,CCVAL)
      ENDDO
      TIME=DTIME(TA)
      PRINT *,'CCGET time.......: ',TIME
C ---
C --- Zero values in compressed column data struture
C ---
      TIME=DTIME(TA)
      CALL CCZERO (ICCPTR,NCOL,CCVAL)
      TIME=DTIME(TA)
      PRINT *,'CCZERO time......: ',TIME
C ---
C --- Add to data in compressed column data structure with CCADD
C ---
      TIME=DTIME(TA)
      DO I=1,NE
        A=CCADD (IROW(I),ICOL(I),VAL(I),ICCPTR,ICCROW,CCVAL,NCOL)
      ENDDO
      TIME=DTIME(TA)
      PRINT *,'CCADD time.......: ',TIME
C ---
C --- End of tests
C ---
      ENDDO
      END
```

A1.10. cczero.f

```
C-----------------------------------------------------------------------
C   cczero.f
C
C   03/13/2005  Rodney Jacobs
C
C   Set matrix element values of a matrix stored in compressed column
C   format to zero.
C
C   The non-zero elements in the FEM matrix have the same row and column
C   indices from iteration to iteration.  Overhead is reduced by saving
C   the compressed column pointers and the row indices between
C   iterations and simply zeroing the matrix element values.
C
C   Input:
C     ICCPTR     compressed column pointers
C     NCOL       number of columns in matrix
C     CCVAL      matrix element values in compressed column format
C
C   Side effects:
C     CCVAL matrix elements are set to zero
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE CCZERO (ICCPTR, NCOL, CCVAL)
      IMPLICIT REAL*8 (A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),CCVAL(MAXNZ)
C
      DO I=1,NCOL
        J1=ICCPTR(I,1)
        J2=ICCPTR(I,2)
        DO WHILE (J1.LE.J2)
          CCVAL(J1)=0.D0
          J1=J1+1
        ENDDO
      ENDDO
      END
```

A1.11. matdump.c

```
/* matdump.c
 *
 *     Dump coefficients and righthand side of Ax = b from compressed
 *     column data structure to a file.
 *
 *     This routine uses the same calling parameters as umfsolve_ and
 *     slusolve_.  It can be inserted in place of either of these routines
 *     to capture a disk file of the equations to be solved.
 *
 *     Input:
 *       iccptr[]    compressed column pointers
 *       iccrow[]    compressed column row indices
 *       cca[]       compressed column matrix elements of A
 *       b[]         righthand side of Ax = b
 *       x[]         solution vector
 *       ncol        number of rows/columns of A
 *       nz          number of elements in iccrow[] and cca[]
 *       base        index base for iccptr[], cca[], b[], and x[]
 *                     0 = base 0 (C, C++)
 *       mode        operation mode
 *                     1 = dump A and b arrays (solve)
 *                     2 = do nothing (deallocate memory)
 *       debug             debug messages
 *                     0 = do not print messages
 *                     1 = print debugging messages
 *
 *     Returns:
 *       0 = OK
 *       1 = Error
 *
 *     Side effects:
 *       Creates output file with following format:
 *            <nz> <ncol>
 *            <i> <j> <a[i],[j]>
 *            <i> <b[i]>
 *
 *       Output filename is matrix# where "#" is one for the first call
 *       and increments by one for each subsequent call.
 *
 *     Rodney Jacobs, University of Maine, 2005
 */

#include <stdio.h>
#include <stdlib.h>

int matdump_ (int iccptr[], int iccrow[], double cca[], double b[],
    double x[], int *ncol, int *nz, int *base, int *mode, int *debug) {

    static int      fileno = 0;
    char   filenm[20];
    FILE   *fp;
    int    col,i;

    if (*mode==2) return 0;

    ++fileno;
    sprintf(filenm,"matrix%i",fileno);
    if (*debug) printf("MATDUMP: dump A and b to file %s\n",filenm);

    if ((fp=fopen(filenm,"w"))==NULL) {
       printf("MATDUMP: open for output failure on %s\n",filenm);
```

```
        return 1;
    }

    if (*base) ccbase0 (*nz,*ncol,iccptr,iccrow);

    fprintf(fp,"%d %d\n",*nz,*ncol);

    for (col=0; col<*ncol; col++) {
        for (i=iccptr[col]; i<iccptr[col+1]; i++) {
            fprintf(fp,"%d %d %lg\n",iccrow[i]+1,col+1,cca[i]);
        }
    }

    for (i=0; i<*ncol; i++) {
        fprintf(fp,"%d %lg\n",i+1,b[i]);
    }

    if (*base) ccbase1 (*nz,*ncol,iccptr,iccrow);

    fclose(fp);
    return 0;
}
```

Appendix 2. Error Measures Routine

```
C-----------------------------------------------------------------------
C   ccberr.f
C
C   Compute backward error of Ax=b where A is in compressed column
C   format.
C
C   Input:
C     ICCPTR
C     ICCROW
C     CCA
C     B          Right hand side of Ax=b
C     X          Vector x of Ax=b
C     NCOL       Number of columns or rows
C     R          Residual vector of Ax-b
C     R2         2-norm of R
C     BERR       Max over I (R(I)/S(I))
C                  S(I)=Sum on J ( ABS(A(I,J)*X(J) ) + ABS(B(I))
C     RINF       Infinity norm of R
C     AINF       Infinity norm of A
C     XINF       Infinity norm of X
C     HINF       Infinity norm of H
C     LDISP      .TRUE. => Display results
C
C   Side effects:
C     R, R2, BERR, RINF, AINF, XINF, and HINF are updated
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE CCBERR (ICCPTR,ICCROW,CCA,B,X,NCOL,R,R2,BERR,RINF,
     &  AINF,XINF,HINF,LDISP)
      IMPLICIT REAL*8(A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCA(MAXNZ),B(MAXCOL)
      DIMENSION X(MAXCOL)
      DIMENSION R(MAXCOL),S(MAXCOL),AINF0(MAXCOL)
      LOGICAL LDISP
C ---
C --- Compute error measures
C ---
      DO I=1,NCOL
        R(I)=-B(I)
        S(I)=ABS(B(I))
        AINF0(I)=0.D0
      ENDDO
C ---
      DO J=1,NCOL
        DO K=ICCPTR(J,1),ICCPTR(J+1,1)-1
          I=ICCROW(K)
          TERM=CCA(K)*X(J)
          R(I)=R(I)+TERM
          S(I)=S(I)+ABS(TERM)
          AINF0(I)=AINF0(I)+ABS(CCA(K))
        ENDDO
      ENDDO
C ---
      BERR=0.D0
      RINF=0.D0
      XINF=0.D0
      AINF=0.D0
      BINF=0.D0
```

```
      DO I=1,NCOL
        BERR0=ABS(R(I)/S(I))
        IF (BERR0.GT.BERR) BERR=BERR0
        IF (ABS(R(I)).GT.RINF) RINF=ABS(R(I))
        IF (ABS(X(I)).GT.XINF) XINF=ABS(X(I))
        IF (ABS(AINF0(I)).GT.AINF) AINF=ABS(AINF0(I))
        IF (ABS(B(I)).GT.BINF) BINF=ABS(B(I))
      ENDDO
      HINF=RINF/(AINF*XINF)
C ---
C --- Display error measures
C ---
      IF (.NOT. LDISP) RETURN
      PRINT *,' '
      PRINT *,'Error Measures'
      PRINT *,'-------------------------------------------'
      PRINT *,'BERR.......................: ',BERR
      PRINT *,'||R||infinity..............: ',RINF
      PRINT *,'||H||infinity (lower bound): ',HINF
      PRINT *,'||A||infinity..............: ',AINF
      PRINT *,'||B||infinity..............: ',BINF
      PRINT *,'||X||infinity..............: ',XINF
      PRINT *,' '
      RETURN
      END
```

Appendix 3. SuperLU Interface Routine and Demonstration Program

A3.1. demo1.f

```
C-----------------------------------------------------------------------
C   demo1.f
C
C   Load and solve the set of linear equations Ax=b using SuperLU
C
C-----------------------------------------------------------------------
      IMPLICIT REAL*8(A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCA(MAXNZ),B(MAXCOL),
     1          X(MAXCOL),R(MAXCOL)
      DIMENSION CCA2(MAXNZ),B2(MAXCOL)
      REAL DTIME,TA(2)
      PRINT *,'SuperLU Demonstration'
      PRINT *,'Rodney Jacobs, University of Maine, 2005'
      PRINT *,' '
C ---
C --- Initialize compressed column data structure
C ---
      PRINT *,'Initialize compressed column data structures'
      OPEN (1,FILE='matrix')
      READ (1,*) NE,NCOL
      PRINT *,'# Non-zero Elements........: ',NE
      PRINT *,'# Columns..................: ',NCOL
      NBKTSZ=81
      CALL CCINIT (ICCPTR,NCOL,NBKTSZ)
      NZ=0
C ---
C --- Read matrix element values and store in compressed column format
C ---
      PRINT *,'Read and store matrix A'
      DO I=1,NE
        READ (1,*) IROW,ICOL,VAL
        CALL CCPUT (IROW,ICOL,VAL,ICCPTR,ICCROW,CCA,NCOL)
      ENDDO
      CALL CCSQZ (ICCPTR,ICCROW,CCA,NCOL,NZ)
C ---
C --- Read and store righthand side values
C ---
      PRINT *,'Read and store righthand side'
      DO I=1,NCOL
        READ (1,*) IROW,VAL
        B(IROW)=VAL
      ENDDO
C ---
C --- Make copies of A and B for CCBERR
C ---
      DO I=1,NCOL
        B2(I)=B(I)
      ENDDO
      DO I=1,NE
        CCA2(I)=CCA(I)
      ENDDO
C ---
C --- Solve the set of equations
C ---
      PRINT *,'Solve Ax=B'
      IBASE=1
      IMODE=1
```

```
      IDEBUG=1
      TIME=DTIME(TA)
      CALL SLUXSOLVE (ICCPTR,ICCROW,CCA,B,X,NCOL,NE,IBASE,IMODE,
     &  IDEBUG)
      TIME=DTIME(TA)
      PRINT *,'Total CPU time (seconds)...:',TIME
C ---
C --- Print x()
C ---
      IF (.FALSE.) THEN
        DO I=1,NCOL
          PRINT *,'X(',I,') =',X(I)
        ENDDO
      ENDIF
C ---
C --- Compute and print error measures
C ---
      CALL CCBERR (ICCPTR,ICCROW,CCA2,B2,X,NCOL,R,R2,BERR,RINF,AINF,
     &  XINF,HINF,.TRUE.)
C ---
C --- Deallocate Symbolic and Numeric objects and stop
C ---
      PRINT *,'Deallocate memory'
      IMODE=2
      CALL SLUSOLVE (ICCPTR,ICCROW,CCA,B,X,NCOL,NE,IBASE,IMODE,IDEBUG)
      CLOSE (1)
      END
```

A3.2. sluxsolve.c

```
/* sluxsolve.c
 *
 * Solve a set of simultaneous linear equations using SuperLU expert solver
 *
 * iccptr     (input) int*
 *            compressed column pointers
 *
 * iccrow     (input) int*
 *            compressed column row indices
 *
 * cca        (input/output) double*
 *            compressed column matrix elements of A
 *               values change when options.Equil=YES
 *
 * b          (input/output) double*
 *            righthand side of Ax = b
 *               values change when options.Equil=YES
 *
 * x          (output) double*
 *            solution vector
 *
 * ncol             (input) int*
 *            number of rows/columns of A
 *
 * nz         (input) int*
 *            number of elements in iccrow[] and cca[]
 *
 * base             (input) int*
 *            index base for iccptr[], iccrow[], cca[], b[], and x[]
 *              0 = base 0 (C, C++)
 *              1 = base 1 (Fortran)
 *
 * mode             (input) int*
 *            mode of operation
 *              1 = solve Ax = b
 *              2 = end of program clean up: deallocate memory
 *
 * debug      (input) int*
 *            debug messages
 *              0 = do not print debugging messages
 *              1 = print debugging messages
 *
 * Returns:   (output) int
 *              0 = OK
 *              1 = Error
 *
 *     Rodney Jacobs, University of Maine, 2005
 */

#include <stdio.h>
#include <math.h>
#include "dsp_defs.h"

double  wallclock();
void    cbase0(int,int,int*,int*);
void    cbase1(int,int,int*,int*);

int sluxsolve_ (int iccptr[], int iccrow[], double cca[], double b[],
    double x[], int *ncol, int *nz, int *base, int *mode, int *debug) {

    static superlu_options_t     options;
```

```
    static int      *etree;
    static int      *perm_r;
    static int      *perm_c;
    static double   *R;
    static double   *C;
    static int      pass = 0;

    SuperMatrix     As, Bs, Xs, Ls, Us;
    char            equed[1];
    void            *work;
    int             lwork;
    int             info;
    SuperLUStat_t   stat;
    double          ferr, berr;
    double          rpg, rcond;
    mem_usage_t     mem_usage;
    double          t0, t1;
    double          *utime;
    flops_t         *ops;

    if (*mode == 1) {

/*-----------------------------------------------------------------------
 * SuperLU initialization
 *-----------------------------------------------------------------------
 */

       if (pass==0) {
           if (*debug) {
              puts("SLUXSOLVE: SuperLU Expert Solver initialization");
           }

           /* Set SuperLU default options
              options.Fact = DOFACT;
              options.Equil = YES;
              options.ColPerm = COLAMD;
              options.Trans = NOTRANS;
              options.IterRefine = NOREFINE;
              options.PrintStat = NO;
              options.SymmetricMode = NO;
              options.DiagPivotThresh = 1.0;
              options.PivotGrowth = NO;
              options.ConditionNumber = NO;
            */
           set_default_options(&options);

           // Set specific options
           options.Equil=YES;
           options.ColPerm=NATURAL;
           options.DiagPivotThresh=0.1;
           options.IterRefine=DOUBLE;
           options.PrintStat=YES;
           options.PivotGrowth=YES;
           options.ConditionNumber=YES;

           // Print options chosen
            if (*debug) {
              t0=wallclock();
              puts("");

              if (options.Fact==DOFACT)
                  puts("Fact=DOFACT.........: Factor matrix A from scratch");
              else if (options.Fact==SamePattern)
```

```
                puts("Fact=SamePattern....: Reuse last column permutation
vector");
            else if (options.Fact==SamePattern_SameRowPerm)
                puts("Fact=SamePattern_SameRowPerm: Reuse last row & column
permutation & scaling");
            else if (options.Fact==FACTORED)
                puts("Fact=FACTORED.......: L, U, perm_r, and perm_c contain
factored form of A");
            else
                puts("Fact=?invalid?");

            if (options.Equil==NO)
                puts("Equil=NO............: Do not scale A");
            else if (options.Equil==YES)
                puts("Equil=YES...........: Scale A's rows and columns to have
unit norm");
            else
                puts("Equil=?invalid?");

            if (options.ColPerm==NATURAL)
                puts("ColPerm=NATURAL.....: Use natural column ordering");
            else if (options.ColPerm==MMD_ATA)
                puts("ColPerm=MMD_ATA.....: Use minimum degree column ordering
on A'*A");
            else if (options.ColPerm==MMD_AT_PLUS_A)
                  puts("ColPerm=MMD_AT_PLUSA: Use minimum degree column
ordering on A'+A");
            else if (options.ColPerm==COLAMD)
                puts("ColPerm=COLAMD......: Use approximate minimum degree
column ordering");
            else if (options.ColPerm==MY_PERMC)
                puts("ColPerm=MU_PERMC....: Use column order specified in
ScalePermstruct->perm_c");
            else
                puts("ColPerm=?invalid?");

            if (options.Trans==NOTRANS)
                puts("Trans=NOTRANS.......: Solve A * X = B (A is not
transposed)");
            else if (options.Trans==TRANS)
                puts("Trans=TRANS.........: Solve A**T * X = B (A is
transposed)");
            else if (options.Trans==CONJ)
                puts("Trans=CONJ..........: Solve A**H * X = B (A is
transposed and conjugated)");
            else
                puts("Trans=?invalid?");

            if (options.IterRefine==NO)
                puts("IterRefine=NO.......: Do not perform iterative
refinement");
            else if (options.IterRefine==SINGLE)
                puts("IterRefine=SINGLE...: Perform single precision iterative
refinement");
            else if (options.IterRefine==DOUBLE)
                puts("IterRefine=DOUBLE...: Perform double precision iterative
refinement");
            else if (options.IterRefine==EXTRA)
                puts("IterRefine=EXTRA....: Perform iterative refinement in
extra precision");
            else
                puts("IterRefine=?invalid?");
```

```
            if (options.PrintStat==NO)
                puts("PrintStat=NO........: Do not print solver's
statistics");
            else if (options.PrintStat==YES)
                puts("PrintStat=YES.......: Print solver's statistics");
            else
                puts("PrintStat=?invalid?");

            if (options.SymmetricMode==NO)
                puts("SymmetricMode=NO....: Assume A is not diagonally
dominant");
            else if (options.SymmetricMode==YES)
                puts("SymmetricMode=YES...: Assume A is diagonally dominant or
nearly so");
            else
                puts("SymmetricMode=?invalid?");

            printf("Diag Pivot Threshold: %e\n",options.DiagPivotThresh);

            if (options.PivotGrowth==NO)
                puts("PivotGrowth=NO......: Do not compute reciprocal of pivot
growth");
            else if (options.PivotGrowth==YES)
                puts("PivotGrowth=YES.....: Compute reciprocal of pivot
growth");
            else
                puts("PivotGrowth=?invalid?");

            if (options.ConditionNumber==NO)
                puts("ConditionNumber=NO..: Do not compute reciprocal of
condition number");
            else if (options.ConditionNumber==YES)
                puts("ConditionNumber=YES.: Compute reciprocal of condition
number");
            else
                puts("ConditionNumber=?invalid?");

            puts("");
         }

         // Allocate memory
         if (!(etree=intMalloc(*ncol)))
            ABORT("SLUXSOLVE: Maloc fails for etree[].")
         if (!(perm_r=intMalloc(*ncol)))
            ABORT("SLUXSOLVE: Malloc fails for perm_r[].");
         if (!(perm_c=intMalloc(*ncol)))
            ABORT("SLUXSOLVE: Malloc fails for perm_c[].");
         if (!(R=(double *) SUPERLU_MALLOC(*ncol*sizeof(double))))
            ABORT("SLUXSOLVE: SUPERLU_MALLOC fails for R[].");
         if (!(C=(double *) SUPERLU_MALLOC(*ncol*sizeof(double))))
            ABORT("SLUXSOLVE: SUPERLU_MALLOC fails for C[].");

         pass=1;
      } // if (pass==0)

/*-----------------------------------------------------------------------
 * Solve Ax = b
 *-----------------------------------------------------------------------
 */

      // Array base conversion from Fortran to C
      if (*base) ccbase0 (*nz,*ncol,iccptr,iccrow);
```

```
// Debug I/O and initialize time stats
if (*debug) {
    puts("SLUXSOLVE: Solve Ax=b");
}

// Create matrix structures
dCreate_CompCol_Matrix(&As,*ncol,*ncol,*nz,cca,iccrow,iccptr,
    SLU_NC,SLU_D,SLU_GE);
dCreate_Dense_Matrix(&Bs,*ncol,1,b,*ncol,SLU_DN,SLU_D,SLU_GE);
dCreate_Dense_Matrix(&Xs,*ncol,1,x,*ncol,SLU_DN,SLU_D,SLU_GE);

// Initialize SuperLU stats
 StatInit(&stat);

// Solve the system, compute condition number and error bounds
lwork=0;
dgssvx(&options,&As,perm_c,perm_r,etree,equed,R,C,&Ls,&Us,
    work,lwork,&Bs,&Xs,&rpg,&rcond,&ferr,&berr,&mem_usage,
    &stat,&info);

// Print statistics
if (info==0 || info==*ncol+1) {
    if (*debug) {
          printf("# columns (rows)............: %d\n",*ncol);
          printf("# non-zero elements.........: %d\n",*nz);
       if (options.PivotGrowth==YES)
          printf("Pivot growth................: %e\n",1./rpg);
       if (options.ConditionNumber==YES)
          printf("Condition number............: %e\n",1./rcond);
       if (options.IterRefine != NOREFINE)
          printf("Iterative Refinement Steps..: %d\n",
              stat.RefineSteps);
          printf("BERR........................: %e\n",berr);
          printf("FERR........................: %e\n",ferr);
          printf("# nonzeros in L.............: %d\n",
              ((SCformat *) Ls.Store)->nnz);
          printf("# nonzeros in U.............: %d\n",
              ((SCformat *) Us.Store)->nnz);
          printf("# nonzeros in L+U...........: %d\n",
              ((SCformat *) Ls.Store)->nnz+
              ((SCformat *) Us.Store)->nnz)-*ncol;
          printf("L\\U memory (MB)............: %.3f\n",
              mem_usage.for_lu/1.048576e6);
          printf("Total memory needed (MB)....: %.3f\n",
              mem_usage.total_needed/1.048576e6);
          printf("# memory explansions........: %d\n",
              mem_usage.expansions);
          utime=stat.utime;
       ops=stat.ops;
       puts("");
       puts ("Timings    Time       FLOPs       MFLOPs/sec");
       puts ("-------  -------  ------------   -----------");
       timing("Factor",utime[FACT],ops[FACT]);
       timing("Solve",utime[SOLVE],ops[SOLVE]);
       timing("Etree",utime[ETREE],ops[ETREE]);
       timing("Equil",utime[EQUIL],ops[EQUIL]);
       timing("Rcond",utime[RCOND],ops[RCOND]);
       timing("Refine",utime[REFINE],ops[REFINE]);
       puts ("         -------  ------------   -----------");
       timing("Total",utime[FACT]+utime[SOLVE]+utime[ETREE]+
           utime[EQUIL]+utime[RCOND]+utime[REFINE],
           ops[FACT]+ops[SOLVE]+ops[ETREE]+ops[EQUIL]+
           ops[RCOND]+ops[REFINE]);
```

```
            puts("");
            t1=wallclock();
                printf("Wall clock time (seconds)...:  %f\n",t1-t0);
        }

      // Unsuccessful
      } else if (info>0) {
          printf("SLUXSOLVE: Estimated memory: %d bytes\n",info-*ncol);
          ABORT("");
      }

      // Deallocate statistics
      StatFree(&stat);

      // Array base conversion from C to Fortran
      if (*base) ccbasel (*nz,*ncol,iccptr,iccrow);
      return 0;

    } // if (*mode=1)

/*-----------------------------------------------------------------------
 * Free dynamic memory
 *-----------------------------------------------------------------------
 */

    if (*mode==2) {
      if (*debug) puts("SLUXSOLVE: free dynamic memory");
      if (pass==1) {
          SUPERLU_FREE(perm_r);
          SUPERLU_FREE(perm_c);
          SUPERLU_FREE(R);
          SUPERLU_FREE(C);
      }
      pass=0;
      return 0;
    }

/*-----------------------------------------------------------------------
 * Unrecognized mode value
 *-----------------------------------------------------------------------
 */

    printf ("SLUXSOLVE: invalid: mode = %d\n",*mode);
    return 1;
}

/*-----------------------------------------------------------------------
 * Report timings
 *-----------------------------------------------------------------------
 */

int timing(char *s,double t,double f) {
    printf("%7s  %7.2f  %10e  %12.2f\n",s,
      t,f,(f/t)*1e-6);
}
```

Appendix 4. UMFPACK Interface Routine and Demonstration Program

A4.1. demo.f

```
C-----------------------------------------------------------------------
C   demo.f
C
C   Load and solve the set of linear equations Ax=b using UMFPACK
C
C-----------------------------------------------------------------------
      IMPLICIT REAL*8(A-H,O-Z)
      include "ccparam.h"
      DIMENSION ICCPTR(MAXCOL,2),ICCROW(MAXNZ),CCA(MAXNZ),B(MAXCOL),
     1          X(MAXCOL),R(MAXCOL)
      REAL DTIME,TA(2)
      PRINT *,'UMFPACK Demonstration'
      PRINT *,'Rodney Jacobs, University of Maine, 2005'
      PRINT *,' '
C ---
C --- Initialize compressed column data structure
C ---
      PRINT *,'Initialize compressed column data structures'
      OPEN (1,FILE='matrix')
      READ (1,*) NE,NCOL
      PRINT *,'# Non-zero Elements........: ',NE
      PRINT *,'# Columns..................: ',NCOL
      NBKTSZ=81
      CALL CCINIT (ICCPTR,NCOL,NBKTSZ)
      NZ=0
C ---
C --- Read matrix element values and store in compressed column format
C ---
      PRINT *,'Read and store matrix A'
      DO I=1,NE
        READ (1,*) IROW,ICOL,VAL
        CALL CCPUT (IROW,ICOL,VAL,ICCPTR,ICCROW,CCA,NCOL)
      ENDDO
      CALL CCSQZ (ICCPTR,ICCROW,CCA,NCOL,NZ)
C ---
C --- Read and store righthand side values
C ---
      PRINT *,'Read and store righthand side'
      DO I=1,NCOL
        READ (1,*) IROW,VAL
        B(IROW)=VAL
      ENDDO
C ---
C --- Solve the set of equations
C ---
      PRINT *,'Solve Ax=B'
      IBASE=1
      IMODE=1
      IDEBUG=1
      TIME=DTIME(TA)
      CALL UMFSOLVE (ICCPTR,ICCROW,CCA,B,X,NCOL,NE,IBASE,IMODE,IDEBUG)
      TIME=DTIME(TA)
      PRINT *,'Total CPU time (seconds)...:',TIME
C ---
C --- Print x()
C ---
      IF (.FALSE.) THEN
        DO I=1,NCOL
```

```
          PRINT *,'X(',I,') =',X(I)
        ENDDO
      ENDIF
C ---
C --- Compute and print error measures
C ---
      CALL CCBERR (ICCPTR,ICCROW,CCA,B,X,NCOL,R,R2,BERR,RINF,AINF,XINF,
     &  HINF,.TRUE.)
C ---
C --- Deallocate Symbolic and Numeric objects and stop
C ---
      PRINT *,'Deallocate memory'
      IMODE=2
      CALL UMFSOLVE (ICCPTR,ICCROW,CCA,B,X,NCOL,NE,IBASE,IMODE,IDEBUG)
      CLOSE (1)
      END
```

A4.2. umfsolve.c

```
/* umfsolve.c
 *
 *     Solves a set of simultaneous linear equations using UMFPACK.
 *
 *     See UMFPACK/Demo/umf4.c for guidance.
 *
 *     Input:
 *       iccptr[]    compressed column pointers
 *       iccrow[]    compressed column row indices
 *       cca[]       compressed column matrix elements of A
 *       b[]         righthand side of Ax = b
 *       x[]         solution vector
 *       ncol        number of rows/columns of A
 *       nz          number of elements in iccrow[] and cca[]
 *      base  index base for iccptr[], iccrow[], cca[], b[], and x[]
 *                    0 = base 0 (C, C++)
 *                    1 = base 1 (Fortran)
 *       mode        operation mode
 *                    1 = solve Ax = b
 *                    2 = end of program clean up: deallocate memory
 *                        for Symbolic and Numeric objects
 *       debug             debug messages
 *                    0 = do not print debugging messages
 *                    1 = print debugging messages
 *
 *     Returns:
 *       0 = OK
 *       1 = Error
 *
 *     Side effects:
 *       1.  x[] set to solution of Ax = b
 *       2.  The Symbolic object needs to be created whenever the pattern
 *            of non-zero elements of A changes.  The Symbolic object is
 *           created on the first call of UMFPACK_solve and on every call
 *           that follows a call with mode=2.  Otherwise, the previously
 *           created Symbolic object is used.
 *
 *     Rodney Jacobs, University of Maine, 2005`
 */

#include <stdio.h>
#include <math.h>
#include "umfpack.h"
#include "amd.h"

#define FALSE 0

double wallclock();
void   ccbase0(int,int,int*,int*);
void   ccbase1(int,int,int*,int*);

int umfsolve_ (int iccptr[], int iccrow[], double cca[], double b[],
    double x[],int *ncol, int *nz, int *base, int *mode, int *debug) {

    static double Control[UMFPACK_CONTROL], Info[UMFPACK_INFO];
    static void   *Symbolic, *Numeric;
    static int    pass = 0;

    int    status;
    double t0, t1;
```

```
    if (*mode == 1) {

/*-----------------------------------------------------------------------
 * UMFPACK initialization
 *-----------------------------------------------------------------------
 */

      if (pass==0) {
          if (*debug) {
             puts("UMFSOLVE: initialization");
             printf("   ncol = %d\n",*ncol);
             printf("     nz = %d\n",*nz);
             printf("   base = %d\n",*base);
             printf("   mode = %d\n",*mode);
          }

          // Set UMFPACK control parameters here
          umfpack_di_defaults(Control);
          Control[UMFPACK_PRL]=3;
          Control[UMFPACK_BLOCK_SIZE]=32;
          Control[UMFPACK_STRATEGY]=UMFPACK_STRATEGY_AUTO ;
          // Control[UMFPACK_IRSTEP]=0;

          if (*debug) umfpack_di_report_control (Control) ;

/*-----------------------------------------------------------------------
 * Fortran base 1 to C base 0 conversion
 *-----------------------------------------------------------------------
 */

          if (*base) ccbase0 (*nz,*ncol,iccptr,iccrow);

/*-----------------------------------------------------------------------
 * Symbolic factorization
 *-----------------------------------------------------------------------
 */

          if (*debug) {
             puts("UMFSOLVE: Symbolic factorization");
             t0=wallclock();
          }
          status=umfpack_di_symbolic (*ncol,*ncol,iccptr,iccrow,cca,
             &Symbolic,Control,Info);
          if (*debug) {
             t1=wallclock();
             printf("%f secs for symbolic factorization\n",t1-t0);
          }
          if (status != UMFPACK_OK) {
             umfpack_di_report_status(Control,status);
             printf("umfpack_di_sumbolic failed: %d\n",status);
             if (*base) ccbase1 (*nz,*ncol,iccptr,iccrow);
             return 1;
          }
      } // if (pass == 0)

/*-----------------------------------------------------------------------
 * Numeric factorization
 *-----------------------------------------------------------------------
 */

      if (*debug) {
          puts("UMFSOLVE: Numeric factorization");
          t0=wallclock();
```

```
        }
        if (pass==0)
            pass=1;
        else
            if (*base) ccbase0 (*nz,*ncol,iccptr,iccrow);
        status=umfpack_di_numeric (iccptr,iccrow,cca,Symbolic,&Numeric,
             Control,Info);
        if (*debug) {
            t1=wallclock();
            printf("%f secs for numeric factorization\n",t1-t0);
        }
        if (status < UMFPACK_OK) {
            umfpack_di_report_info(Control,Info);
            umfpack_di_report_status(Control,status);
            printf("umfpack_di_numeric failed: %d\n",status);
            if (*base) ccbase1 (*nz,*ncol,iccptr,iccrow);
            return 1;
        }

/*-----------------------------------------------------------------------
 * Sovle Ax=b
 *-----------------------------------------------------------------------
 */

        if (*debug) {
            puts("UMFSOLVE: Solve Ax=b");
            t0=wallclock();
        }
        status=umfpack_di_solve (UMFPACK_A,iccptr,iccrow,cca,x,b,
            Numeric,Control,Info);
        if (*debug) {
            t1=wallclock();
            printf("%f secs to solve\n",t1-t0);
        }
        if (status < UMFPACK_OK) {
            printf ("umfpack_di_solve failed: %d\n",status);
            if (*base) ccbase1 (*nz,*ncol,iccptr,iccrow);
            return 1;
          }

/*-----------------------------------------------------------------------
 * Free Numeric factorization
 *-----------------------------------------------------------------------
 */

          umfpack_di_free_numeric (&Numeric);

/*-----------------------------------------------------------------------
 * Additional reporting
 *-----------------------------------------------------------------------
 */

        if (*debug) umfpack_di_report_info(Control,Info);
        if (*base) ccbase1 (*nz,*ncol,iccptr,iccrow);
        return 0;
    } // if (*mode == 1)

/*-----------------------------------------------------------------------
 * Free Symbolic factorization
 *-----------------------------------------------------------------------
 */

    if (*mode == 2) {
```

```
      if (*debug) puts("UMFSOLVE: free Symbolic object");
       umfpack_di_free_symbolic (&Symbolic);
      pass=0;
      return 0;
    }

/*-----------------------------------------------------------------------
 * Unrecognized mode value
 *-----------------------------------------------------------------------
 */

    printf ("UMFSOLVE: invalid: mode = %d\n",*mode);
    return 1;
}
```

Appendix 5. Simple Banded Gaussian Elimination Program

```
C-----------------------------------------------------------------------
C   bg0.f
C
C   Load and solve the set of linear equations Ax=b using banded Gaussian
C   elimination.
C
C-----------------------------------------------------------------------
      IMPLICIT REAL*8(A-H,O-Z)
      PARAMETER (MAXROW=24000,MAXBW=1241)
      DIMENSION A(MAXBW,MAXROW),A1(MAXBW,MAXROW),B(MAXROW),B1(MAXROW)
      DIMENSION X(MAXROW)
      SAVE A,A1,B,B1,X
      REAL DTIME,TA(2)
      INTEGER*8 NFLOP1,NFLOP2
C ---
      PRINT *,'Banded Gaussian Elimination (BG0)'
      PRINT *,'   Assume Constant Upper Bandwidth'
      PRINT *,'Rodney Jacobs, University of Maine, 2005'
      PRINT *,' '
C ---
C --- Determine lower and upper bandwidth of data
C ---
      PRINT *,'Bandwidth determination'
      OPEN (1,FILE='matrix')
      READ (1,*) NE,NROW
      PRINT *,'# Non-zero Elements.............: ',NE
      PRINT *,'# Rows..........................: ',NROW
      NBWL=0
      NBWU=0
      DO I=1,NE
        READ (1,*) IROW,ICOL,VAL
        IF (ICOL.LT.IROW) THEN
          IF (IROW-ICOL.GT.NBWL) NBWL=IROW-ICOL
        ELSE
          IF (ICOL-IROW.GT.NBWU) NBWU=ICOL-IROW
        ENDIF
      ENDDO
      CLOSE (1)
      PRINT *,'Lower Bandwidth.................: ',NBWL
      PRINT *,'Upper Bandwidth.................: ',NBWU
      NBWL1=NBWL+1
      IF (NROW.GT.MAXROW) THEN
        PRINT *,'MAXROW = ',MAXROW,' exceeded'
        STOP
      ENDIF
      IF (NBWL1+NBWU.GT.MAXBW) THEN
        PRINT *,'MAXBW = ',MAXBW,' exceeded'
        STOP
      ENDIF
C ---
C --- Read matrix element values and store in banded format
C ---
      PRINT *,'Read and store matrix A'
      DO J=1,NROW
        DO I=1,NBWL1+NBWU
          A(I,J)=0.D0
          A1(I,J)=0.D0
        ENDDO
        B(J)=0.D0
        B1(J)=0.D0
```

```
      ENDDO
      OPEN (1,FILE='matrix')
      READ (1,*) NE,NROW
      DO I=1,NE
        READ (1,*) IROW,ICOL,VAL
        A(ICOL-IROW+NBWL1,IROW)=VAL
        A1(ICOL-IROW+NBWL1,IROW)=VAL
      ENDDO
C ---
C --- Read and store righthand side values
C ---
      PRINT *,'Read and store righthand side'
      DO I=1,NROW
        READ (1,*) IROW,VAL
        B(IROW)=VAL
        B1(IROW)=VAL
      ENDDO
C ---
C --- Reduce A to upper triangular form using Gaussian elimination
C ---
      PRINT *,'Reduce A to upper triangular form'
      NFLOP1=0
      T1=DTIME(TA)
      DO NR=1,NROW
        K=NR+NBWL
        IF (K.GT.NROW) K=NROW
        DO MR=NR+1,K
          XMULT=A(NBWL1-MR+NR,MR)/A(NBWL1,NR)
          DO NC=NBWL1+1,NBWL1+NBWU
            MC=NC-MR+NR
            A(MC,MR)=A(MC,MR)-XMULT*A(NC,NR)
          ENDDO
          B(MR)=B(MR)-XMULT*B(NR)
          NFLOP1=NFLOP1+NBWU+NBWU+3
        ENDDO
      ENDDO
      T1=DTIME(TA)
C ---
C --- Solve for x
C ---
      PRINT *,'Solve for x using backward substitution'
      NFLOP2=0
      DO NR=NROW,1,-1
        XSUM=B(NR)
        K=NR+NBWU
        IF (K.GT.NROW) K=NROW
        DO J=NR+1,K
          NC=J-NR+NBWL1
          XSUM=XSUM-A(NC,NR)*X(J)
        ENDDO
        X(NR)=XSUM/A(NBWL1,NR)
        NFLOP2=NFLOP2+2*(K-NR)+1
      ENDDO
      T2=DTIME(TA)
C ---
C --- Print x
C ---
      IF (.FALSE.) THEN
        DO NR=1,NROW
          PRINT *,'X(',NR,') =',X(NR)
        ENDDO
      ENDIF
C ---
```

```
C --- Report results
C ---
      PRINT *,'Triangular Reduction Time (sec).: ',T1
      PRINT *,'Backward Substitution Time (sec): ',T2
      PRINT *,'Total Solve Time (sec)..........: ',T1+T2
      PRINT *,'Triangular Reduction FLOPS......: ',NFLOP1
      PRINT *,'Backward Substitution FLOPS.....: ',NFLOP2
      PRINT *,'Total Solve FLOPS...............: ',NFLOP1+NFLOP2
C ---
C --- Compute backward error
C ---
      BERR=0.D0
      DO NR=1,NROW
        R=0.D0
        S=0.D0
        NCL=NR-NBWL
        IF (NCL.LT.1) NCL=1
        NCU=NR+NBWU
        IF (NCU.GT.NROW) NCU=NROW
        DO NC=NCL,NCU
          K=NC-NR+NBWL1
          TERM=A1(K,NR)*X(NC)
          R=R+TERM
          S=S+ABS(TERM)
        ENDDO
        R=R-B1(NR)
        S=S+ABS(B1(NR))
        BERR0=ABS(R/S)
        IF (BERR0.GT.BERR) BERR=BERR0
      ENDDO
      PRINT *,'BERR...........................: ',BERR
C ---
C --- End program
C ---
      CLOSE (1)
      END
```

Appendix 6. Banded Gaussian Elimination Routines

A6.1. bandparam.h

```
C-----------------------------------------------------------------------------
C   bandparam.h
C
C   Define prarameters, control values, and statistical information storage
C   for the banded matrix subroutine library.
C
C-----------------------------------------------------------------------------
C ---
C --- MAXROW is the number of rows allocated to store the banded matrix.  It
C --- must be at least as large as the number of rows in the matrix.
C ---
C --- MAXBW is the number of columnms allocated to store the banded matrix.
C --- If partial pivoting is not used, it must be at least as large as the
C --- bandwidth of the the matrix.  If partial pivoting is used, it must
C --- be at least as large as the matrix bandwidth plus the upper bandwidth
C --- of the matrix to allow for row interchanges.
C ---
      PARAMETER (MAXROW=24000,MAXBW=1241)
C ---
C --- BCTL contains the following control values used by the subroutine
C --- library.  Be sure to call BINIT to initialize this array before calling
C --- any other library routines.
C ---
C ---   BCTL(1): Reporting
C ---              0 = Display severe error messages only
C ---              1 = Display warning messages and severe error messages
C ---              2 = Display progress, warning, and severe error messages
C ---              3 = Display statistics and all other messages
C ---   BCTL(2): Print solution vector
C ---              0 = Do not print
C ---              1 = Print
C ---   BCTL(3): Scale matrix before reducing or factoring it
C ---              0 = Do not scale
C ---              1 = Scale each row by sum of absolute values of
C ---                  coefficients and right hand side value in the row
C ---   BCTL(4): Matrix permutations
C ---              0 = Do not perform any matrix permutations
C ---              1 = Partial pivoting by row interchange using BCTL(5)
C ---   BCTL(5): Partial pivoting threshold
C ---            Must satisfy 0 < BCTL(4) <= 1. A potential pivot a(i,j) is
C ---            acceptable if abs(a(i,j)) >= BCTL(5)*abs(a(*,j)).
C ---            If a(i,j) is not an acceptable pivot, then it is interchanged
C ---            with the row having max(abs(a(*,j))).
C ---   BCTL(6): Iterative refinement (For LU factorizations only)
C ---            If positive, perform iterative refinement until there is
C ---              no improvement in the solution, but iterate no more than
C ---              BCTL(6) times.
C ---            If zero, do not perform iterative refinement.
C ---            If less than zero, perform iterative refinement ABS(BCTL(6))
C ---              times.
C ---   BCTL(7): Print solution vector after performing iterative refinement
C ---              0 = Do not print
C ---              1 = Print
C ---
      DIMENSION BCTL(7)
C ---
C --- BINFO contains the following information and statistics.
C ---
```

```
C ---   BINFO(1):  Non-zero elements in matrix.          Set by BLOAD
C ---   BINFO(2):  Number of rows in matrix.             Set by BLOAD
C ---   BINFO(3):  Lower bandwidth of matrix.            Set by BLOAD
C ---   BINFO(4):  Upper bandwidth of matrix.            Set by BLOAD
C ---   BINFO(5):  Gaussian elimination reduction floating point operations.
C ---                                                   Set by BGE
C ---   BINFO(6):  Gaussian elimination reduction wall clock time (seconds).
C ---                                                   Set by BGE
C ---   BINFO(7):  Gaussian elimination substitution floating point operations.
C ---                                                   Set by BGE
C ---   BINFO(8):  Gaussian elimination substitution wall clock time (seconds).
C ---                                                   Set by BGE
C ---   BINFO(9):  LU factorization floating point operations
C ---                                                   Set by BLUFAC
C ---   BINFO(10): LU factorization wall clock time (seconds)
C ---                                                   Set by BLUFAC
C ---   BINFO(11): LU forward+backward substitution floating point operations.
C ---                                                   Set by BLUSOLVE
C ---   BINFO(12): LU forward+backward substitution wall clock time (seconds).
C ---                                                   Set by BLUSOLVE
C ---   BINFO(13): LU factorization: Non-zeros in L excluding diagonal.
C ---              Diagonal elements of L implicitly have value = 1.
C ---                                                   Set by BLUSTATS
C ---   BINFO(14): LU factorization: Non-zeros on diagonal of U.
C ---                                                   Set by BLUSTATS
C ---   BINFO(15): LU factorization: Non-zeros in U exluding diagonal.
C ---                                                   Set by BLUSTATS
C ---   BINFO(16): BERR Backward error.                 Set by BERROR
C ---   BINFO(17): ||R||infinity  R is residual vector  Set by BERROR
C ---   BINFO(18): ||H||infinity (lower bound)  H is perturbation of A to
C ---              produce x as the exact solution of (A+H)x=b.
C ---                                                   Set by BERROR
C ---   BINFO(19): ||A||infinity  A is matrix in Ax=b   Set by BERROR
C ---   BINFO(20): ||B||          B is rhs in Ax=b       Set by BERROR
C ---   BINFO(21): ||X||infinity  X is solution vector  Set by BERROR
C ---   BINFO(22): Number of iterative refinement steps performed
C ---                                                   Set by BREFINE
C ---   BINFO(23): Number of row permuations performed
C ---                                                   Set by BGE
C ---
      DIMENSION BINFO(23)
```

A6.2. bcopy.f

```
C-----------------------------------------------------------------------
C   bcopy.f
C
C   Copy a set of arrays representing a banded matrix to another set of
C   arrays.
C
C   Input:
C     A      Matrix to copy
C     NCOLL  Column index of leftmost non-zero term in row of array A
C     NCOLU  Column index of rightmost non-zero term in row of array A
C     B      Right hand side to copy
C
C   Side effects:
C     A     copied to A1
C     NCOLL copied to NCOLL1
C     NCOLU copied to NCOLU1
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE BCOPY (A,NCOLL,NCOLU,B,A1,NCOLL1,NCOLU1,B1,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),B(MAXROW)
      DIMENSION A1(MAXBW,MAXROW),NCOLL1(MAXROW),NCOLU1(MAXROW),
     &  B1(MAXROW)
C ---
C --- Copy arrays
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BCOPY: Copy arrays representing a banded matrix'
      NROW=BINFO(2)
      NBW=BINFO(3)+1+BINFO(4)
      IF (BCTL(4).EQ.1) NBW=NBW+BINFO(3)
      DO I=1,NROW
        DO J=1,NBW
          A1(J,I)=A(J,I)
        ENDDO
      ENDDO
      DO I=1,NROW
        B1(I)=B(I)
        NCOLL1(I)=NCOLL(I)
        NCOLU1(I)=NCOLU(I)
      ENDDO
      END
```

A6.3. berror.f

```
C-----------------------------------------------------------------------
C   berror.f
C
C   Compute error measures for banded LU factorization and solve
C
C   Input:
C     A       Matrix of Ax=b in banded storage format
C     NCOLL   Column index of leftmost non-zero term in row of array A
C     NCOLU   Column index of rightmost non-zero term in row of array A
C     X       Solution vector of Ax=b
C     B       Right hand side of Ax=b
C     R       Residual vector Ax-b
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     Set error measures in BINFO
C     Compute residual vector R
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE BERROR (A,NCOLL,NCOLU,X,B,R,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),X(MAXROW),
     &  B(MAXROW),R(MAXROW)
C ---
C --- Compute error measures
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BERROR: Compute error measures'
      NROW=BINFO(2)
      NBWL=BINFO(3)
      NBWL1=NBWL+1
      BERR=0.D0
      RINF=0.D0
      XINF=0.D0
      AINF=0.D0
      BINF=0.D0
      DO NR=1,NROW
        R1=-B(NR)
        S1=ABS(B(NR))
        A1=0.D0
        NCL=NR-(NBWL1-NCOLL(NR))
        NCU=NR+NCOLU(NR)-NBWL1
        DO NC=NCL,NCU
          K=NC-NR+NBWL1
          TERM=A(K,NR)*X(NC)
          R1=R1+TERM
          S1=S1+ABS(TERM)
          A1=A1+ABS(A(K,NR))
        ENDDO
        R(NR)=R1
        BERR0=ABS(R1/S1)
        IF (BERR0.GT.BERR) BERR=BERR0
        IF (ABS(R1).GT.RINF) RINF=ABS(R1)
        IF (ABS(X(NR)).GT.XINF) XINF=ABS(X(NR))
        IF (A1.GT.AINF) AINF=A1
        IF (ABS(B(NR)).GT.BINF) BINF=ABS(B(NR))
      ENDDO
      HINF=RINF/(AINF*XINF)
```

```
C ---
C --- Update BINFO error measures
C ---
      BINFO(16)=BERR
      BINFO(17)=RINF
      BINFO(18)=HINF
      BINFO(19)=AINF
      BINFO(20)=BINF
      BINFO(21)=XINF
C ---
C --- Display error measures
C ---
      IF (BCTL(1).GE.3) THEN
        PRINT *,'Error Measures'
        PRINT *,'---------------------------------------------'
        PRINT *,'BERR.......................: ',BERR
        PRINT *,'||R||infinity..............: ',RINF
        PRINT *,'||H||infinity (lower bound): ',HINF
        PRINT *,'||A||infinity..............: ',AINF
        PRINT *,'||B||infinity..............: ',BINF
        PRINT *,'||X||infinity..............: ',XINF
      ENDIF
      END
```

A6.4. bgauss.f

```
C-----------------------------------------------------------------------
C   bgauss.f
C
C   Load and solve the set of linear equations Ax=b using banded
C   Gaussian elimination
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),B(MAXROW)
      DIMENSION A1(MAXBW,MAXROW),NCOLL1(MAXROW),NCOLU1(MAXROW),
     &  B1(MAXROW)
      DIMENSION X(MAXROW),R(MAXROW)
      SAVE A,NCOLL,NCOLU,B,A1,NCOLL1,NCOLU1,B1,X,R
C ---
C --- Initialization
C ---
      PRINT *,'Banded Gaussian Elimination'
      PRINT *,'Rodney Jacobs, University of Maine, 2005'
      CALL BINIT (BCTL,BINFO)
      CALL BUSER (BCTL,BINFO)
C ---
C --- Read and scale data and load arrays.
C ---
      CALL BLOAD (A,NCOLL,NCOLU,B,BCTL,BINFO)
      CALL BCOPY (A,NCOLL,NCOLU,B,A1,NCOLL1,NCOLU1,B1,BCTL,BINFO)
      CALL BSCALE (A,NCOLL,NCOLU,B,BCTL,BINFO)
C ---
C --- Reduce and solve Ax=b
C ---
      CALL BGE (A,NCOLL,NCOLU,X,B,BCTL,BINFO)
C ---
C --- Print x.  This code was moved from bge.f because of its detrimental
C --- affect on performance.
C ---
      IF (BCTL(2).EQ.1) THEN
        DO I=1,BINFO(2)
          PRINT *,'X(',I,') =',X(I)
        ENDDO
      ENDIF
C ---
C --- Compute error measures and residual vector R
C ---
      CALL BERROR (A1,NCOLL1,NCOLU1,X,B1,R,BCTL,BINFO)
      END
```

A6.5. bge.f

```
C-----------------------------------------------------------------------
C   bge.f
C
C   Reduce and solve a set of linear equations Ax=b using banded Gaussian
C   elimination.
C
C   Input:
C     A       Matrix of Ax=b in banded storage format
C     NCOLL   Column index of leftmost non-zero term in row of array A
C     NCOLU   Column index of rightmost non-zero term in row of array A
C     B       Right hand side of Ax=b
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     Set A to row echolon form
C     NCOLU may be updated to reflect the changed contents of A
C     Set X to solution of Ax=b
C     Set FLOP count and execution time in BINFO
C     Set count of row interchanges in BINFO
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE BGE (A,NCOLL,NCOLU,X,B,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),B(MAXROW),
     &  X(MAXROW)
      DIMENSION ATEMP(MAXBW)
      REAL DTIME,TA(2)
C ---
C --- Reduce A to upper triangular form using Gaussian elimination
C ---
      IF (BCTL(1).GE.2) THEN
        PRINT *,'BGE: Reduce A to upper triangular form'
        IF (BCTL(3).EQ.1)
     &    PRINT *,'BGE: Partial pivoting enabled'
      ENDIF
      NROW=BINFO(2)
      NBWL=BINFO(3)
      NBWL1=NBWL+1
      FLOP=0.D0
      BINFO(23)=0.D0
      TIME=DTIME(TA)
      DO NR=1,NROW
C --- Compute upper limit of rows affected
        K=NR+NBWL
        IF (K.GT.NROW) K=NROW
C --- Pivot test
        IF (BCTL(4).EQ.1) THEN
          PABS=ABS(A(NBWL1,NR))
          NPROW=NR
          DO MR=NR+1,K
            MC=NBWL1-MR+NR
            IF (NPROW.EQ.NR) THEN
              IF (BCTL(5)*ABS(A(MC,MR)).GT.PABS) THEN
                PABS=ABS(A(MC,MR))
                NPROW=MR
              ENDIF
            ELSE
              IF (ABS(A(MC,MR)).GT.PABS) THEN
```

```
                PABS=ABS(A(MC,MR))
                NPROW=MR
              ENDIF
            ENDIF
          ENDDO
C --- Swap rows if the current row does not contain the chosen pivot
          IF (NPROW.NE.NR) THEN
            DO I=NBWL1,NCOLU(NR)
              ATEMP(I)=A(I,NR)
            ENDDO
            NDELTA=NPROW-NR
            DO I=NBWL1-NDELTA,NCOLU(NPROW)
              A(I+NDELTA,NR)=A(I,NPROW)
            ENDDO
            DO I=NCOLU(NPROW)+NDELTA+1,NCOLU(NR)
              A(I,NR)=0.D0
            ENDDO
            DO I=NBWL1,NCOLU(NR)
              A(I-NDELTA,NPROW)=ATEMP(I)
            ENDDO
            DO I=NCOLU(NR)-NDELTA+1,NCOLU(NPROW)
              A(I,NPROW)=0.D0
            ENDDO
            BTEMP=B(NR)
            B(NR)=B(NPROW)
            B(NPROW)=BTEMP
            NTEMP=NCOLU(NR)
            NCOLU(NR)=NCOLU(NPROW)+NDELTA
            NCOLU(NPROW)=NTEMP-NDELTA
            BINFO(23)=BINFO(23)+1.D0
          ENDIF
C --- End pivot test
        ENDIF
C --- Check for zero pivot
        PIVOT=A(NBWL1,NR)
        IF (PIVOT.EQ.0.D0) THEN
          PRINT *,'BGE: Pivot = 0.D0'
          STOP
        ENDIF
C --- Elimination step
        DO MR=NR+1,K
          XMULT=A(NBWL1-MR+NR,MR)
          IF (XMULT.NE.0.D0) THEN
            XMULT=XMULT/PIVOT
            MC=0
            DO NC=NBWL1+1,NCOLU(NR)
              MC=NC-MR+NR
              A(MC,MR)=A(MC,MR)-XMULT*A(NC,NR)
            ENDDO
            B(MR)=B(MR)-XMULT*B(NR)
            IF (NCOLU(MR).LT.MC) NCOLU(MR)=MC
            NCNT=NCOLU(NR)-NBWL1
            IF (NCNT.LT.0) NCNT=0
            FLOP=FLOP+2*NCNT+3
          ENDIF
        ENDDO
      ENDDO
      BINFO(5)=FLOP
      BINFO(6)=DTIME(TA)
C ---
C --- Solve for x
C ---
      IF (BCTL(1).GE.2)
```

```
     &  PRINT *,'BGE: Solve for x using backward substitution'
      FLOP=0
      DO NR=NROW,1,-1
        XSUM=B(NR)
        K=NR+(NCOLU(NR)-NBWL1)
        DO J=NR+1,K
          NC=J-NR+NBWL1
          XSUM=XSUM-A(NC,NR)*X(J)
        ENDDO
        X(NR)=XSUM/A(NBWL1,NR)
        NCNT=K-NR
        IF (NCNT.LT.0) NCNT=0
        FLOP=FLOP+2*(NCNT)+1
      ENDDO
      BINFO(7)=FLOP
      BINFO(8)=DTIME(TA)
C ---
C --- Print x
C ---
c      IF (BCTL(2).EQ.1) THEN
c        DO NR=1,NROW
C --- The following print line reduces the performance of the reduction
C --- loop by approximately 16%!
c          PRINT *,'X(',NR,') =',X(NR)
c        ENDDO
c      ENDIF
C ---
C --- Report results
C ---
      IF (BCTL(1).GE.3) THEN
        PRINT *,'Gaussian Elimination Statistics'
        PRINT *,'-----------------------------------------'
        IF (BCTL(4).EQ.1) THEN
          PRINT *,'Row interchange threshold..: ',BCTL(5)
          PRINT *,'Row interchanges...........: ',BINFO(23)
        ELSE
          PRINT *,'No row interchanges'
        ENDIF
        PRINT *,'Reduction Time (sec).......: ',BINFO(6)
        PRINT *,'Substitution Time (sec)....: ',BINFO(8)
        PRINT *,'Total Solve Time (sec).....: ',BINFO(6)+BINFO(8)
        PRINT *,'Reduction FLOPS............: ',BINFO(5)
        PRINT *,'Substitution FLOPS.........: ',BINFO(7)
        PRINT *,'Total Solve FLOPS..........: ',BINFO(5)+BINFO(7)
      ENDIF
      END
```

A6.6. binit.f

```
C-----------------------------------------------------------------------------
C   binit.f
C
C   Initialize BCTL and BINFO arrays.  See bandparam.h for definitions.
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------------
      SUBROUTINE BINIT (BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
C --- Reporting and progress messages
      BCTL(1)=3
C --- Print solution vector
      BCTL(2)=0
C --- Scale matrix before reducing or factoring
      BCTL(3)=0
C --- Matrix permutation method
      BCTL(4)=0
C --- Partial pivoting threshold
      BCTL(5)=0.1D0
C --- Iterative refinement
      BCTL(6)=0
C --- Print solution vector at each step of iterative refinement
      BCTL(7)=0
C --- Initialize statistics
      DO I=1,23
        BINFO(I)=0.D0
      ENDDO
      END
```

A6.7. bload.f

```
C-----------------------------------------------------------------------
C   bload.f
C
C   Read and load banded matrix A and right hand side b into arrays from
C   data file named "matrix".
C
C   Input:
C     BCTL      See bandparam.h
C
C   Side effects: The following variables are updated:
C     A       Matrix A in Ax=b.  Banded storage format is used.
C     NCOLL   Column index of leftmost non-zero term in row of array A
C     NCOLU   Column index of rightmost non-zero term in row of array A
C     B       Right hand side in Ax=b
C     BINFO   See bandparam.h
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE BLOAD (A,NCOLL,NCOLU,B,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),B(MAXROW),NCOLL(MAXROW),NCOLU(MAXROW)
C ---
C --- Determine lower and upper bandwidth of matrix
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLOAD: Bandwidth determination'
      OPEN (1,FILE='matrix')
      READ (1,*) NE,NROW
      NBWL=0
      NBWU=0
      DO I=1,NE
        READ (1,*) IROW,ICOL,VAL
        IF (ICOL.LT.IROW) THEN
          IF (IROW-ICOL.GT.NBWL) NBWL=IROW-ICOL
        ELSE
          IF (ICOL-IROW.GT.NBWU) NBWU=ICOL-IROW
        ENDIF
      ENDDO
      CLOSE (1)
      IF (BCTL(1).GE.3) THEN
        PRINT *,'BLOAD: # Non-zero Elements.: ',NE
        PRINT *,'BLOAD: # Rows..............: ',NROW
        PRINT *,'BLOAD: Lower Bandwidth.....: ',NBWL
        PRINT *,'BLOAD: Upper Bandwidth.....: ',NBWU
        PRINT *,'BLOAD: MAXROW..............: ',MAXROW
        PRINT *,'BLOAD: MAXBW...............: ',MAXBW
      ENDIF
      IF (NROW.GT.MAXROW) THEN
        PRINT *,'BLOAD: MAXROW = ',MAXROW,' exceeded'
        STOP
      ENDIF
      NBW=NBWL+1+NBWU
      IF (BCTL(4).EQ.1) THEN
        NBW=NBW+NBWL
        IF (BCTL(1).GE.1) PRINT *,'BLOAD: Bandwidth required..: ',NBW,
     &    '  (Partial pivoting row interchanges)'
      ENDIF
      IF (NBW.GT.MAXBW) THEN
        PRINT *,'BLOAD: MAXBW = ',MAXBW,' exceeded'
        STOP
```

```
      ENDIF
      BINFO(1)=NE
      BINFO(2)=NROW
      BINFO(3)=NBWL
      BINFO(4)=NBWU
      NBWL1=NBWL+1
C ---
C --- Initialize arrays
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLOAD: Initialize arrays'
      DO J=1,NROW
        DO I=1,NBWL1+NBWU
          A(I,J)=0.D0
        ENDDO
        B(J)=0.D0
        NCOLL(J)=NBWL1
        NCOLU(J)=NBWL1
      ENDDO
C ---
C --- Read and store matrix element values in banded format
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLOAD: Read and store matrix A'
      OPEN (1,FILE='matrix')
      READ (1,*) NE,NROW
      DO I=1,NE
        READ (1,*) IROW,ICOL,VAL
        NC=ICOL-IROW+NBWL1
        A(NC,IROW)=VAL
        IF (NC.LT.NCOLL(IROW)) NCOLL(IROW)=NC
        IF (NC.GT.NCOLU(IROW)) NCOLU(IROW)=NC
      ENDDO
C ---
C --- Read and store righthand side values
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLOAD: Read and store righthand side'
      DO I=1,NROW
        READ (1,*) IROW,VAL
        B(IROW)=VAL
      ENDDO
C ---
C --- End of routine
C ---
      CLOSE (1)
      END
```

A6.8. bscale.f

```
C-----------------------------------------------------------------------------
C   bscale.f
C
C   Perform LU factorization of a banded matrix A without partial pivoting.
C
C   Input:
C     A       Matrix to be factored in banded storage format
C     NCOLL   Column index of leftmost non-zero term in row of array A
C     NCOLU   Column index of rightmost non-zero term in row of array A
C     B       Right hand side of Ax=b
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     For each row, A and B are scaled by the sum of the absolute values of
C       the coefficients of A and the absolute value of B in that row.
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------------
      SUBROUTINE BSCALE (A,NCOLL,NCOLU,B,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),B(MAXROW)
      REAL DTIME,TA(2)
C ---
C --- Scale rows
C ---
      IF (BCTL(3).EQ.1) THEN
        NROW=BINFO(2)
        IF (BCTL(1).GE.2)
     &    PRINT *,'BSCALE: Scale rows of A'
        DO I=1,NROW
          XSUM=ABS(B(I))
          DO J=NCOLL(I),NCOLU(I)
            XSUM=XSUM+ABS(A(J,I))
          ENDDO
          DO J=NCOLL(I),NCOLU(I)
            A(J,I)=A(J,I)/XSUM
          ENDDO
          B(I)=B(I)/XSUM
        ENDDO
      ENDIF
      END
```

A6.9. buser.f

```
C-----------------------------------------------------------------------------
C   buser.f
C
C   Let the user specify BCTL parameter values.
C
C   Input:
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     Updates BCTL and BINFO
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------------
      SUBROUTINE BUSER (BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      LOGICAL DONE
      DONE=.FALSE.
      DO WHILE (.NOT. DONE)
        I=-1
        DO WHILE (I.NE.0)
          CALL BDISPLAY (BCTL)
          I=-1
          DO WHILE (I.LT.0 .OR. I.GT.7)
            PRINT *,'Enter # to change or 0:'
            READ *,I
          ENDDO
          IF (I.EQ.1) THEN
            CALL BCTL1 (BCTL(1))
          ELSE IF (I.EQ.2) THEN
            CALL BCTL2 (BCTL(2))
          ELSE IF (I.EQ.3) THEN
            CALL BCTL3 (BCTL(3))
          ELSE IF (I.EQ.4) THEN
            CALL BCTL4 (BCTL(4))
          ELSE IF (I.EQ.5) THEN
            CALL BCTL5 (BCTL(5))
          ELSE IF (I.EQ.6) THEN
            CALL BCTL6 (BCTL(6))
          ELSE IF (I.EQ.7) THEN
            CALL BCTL7 (BCTL(7))
          ENDIF
        ENDDO
        CALL BVALID (DONE,BCTL)
      ENDDO
      DO I=1,23
        BINFO(I)=0.D0
      ENDDO
      END
C -----------------------------------------------------------------------------
C ---
C --- BDISPLAY: Display BCTL settings
C ---
      SUBROUTINE BDISPLAY (BCTL)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      PRINT *,' '
      PRINT *,'Control Settings'
      PRINT *,'-----------------------------------------------------------
     &----------------'
C ---
```

```
      IF (BCTL(1).EQ.0) THEN
        PRINT *,' 1. Display only severe errors'
      ELSE IF (BCTL(1).EQ.1) THEN
        PRINT *,' 1. Display severe errors and warning messages'
      ELSE IF (BCTL(1).EQ.2) THEN
        PRINT *,' 1. Display severe errors, warnings, and progress messa
     &ages'
      ELSE IF (BCTL(1).EQ.3) THEN
        PRINT *,' 1. Display all messages plus statistics'
      ELSE
        PRINT *,' 1. Invalid value!'
      ENDIF
C---
      IF (BCTL(2).EQ.0) THEN
        PRINT *,' 2. Do not print solution vector'
      ELSE IF (BCTL(2).EQ.1) THEN
        PRINT *,' 2. Print solution vector'
      ELSE
        PRINT *,' 2. Invalid value!'
      ENDIF
C ---
      IF (BCTL(3).EQ.0) THEN
        PRINT *,' 3. Do not scale matrix'
      ELSE IF (BCTL(3).EQ.1) THEN
        PRINT *,' 3. Scale matrix'
      ELSE
        PRINT *,' 3. Invalid value!'
      ENDIF
C ---
      IF (BCTL(4).EQ.0) THEN
        PRINT *,' 4. Do not perform any matrix permutations'
      ELSE IF (BCTL(4).EQ.1) THEN
        PRINT *,' 4. Perform partial pivoting by interchanging rows'
      ELSE
        PRINT *,' 4. Invalid value!'
      ENDIF
C ---
      IF (BCTL(5).GE.0.D0 .AND. BCTL(5).LE.1.D0) THEN
        PRINT *,' 5. Partial pivoting threshold =',BCTL(5)
      ELSE
        PRINT *,' 5. Invalid value!'
      ENDIF
C ---
      IF (BCTL(6).EQ.0) THEN
        PRINT *,' 6. Do not perform iterative refinement with LU factori
     &zation'
      ELSE IF (BCTL(6).LT.0) THEN
        PRINT *,' 6. Perform',ABS(BCTL(6)),' iterative refinement steps
     &with LU factorization'
      ELSE
        PRINT *,' 6. Perform at most',BCTL(6),' iterative refinement ste
     &ps with LU factorization'
      ENDIF
C ---
      IF (BCTL(7).EQ.0) THEN
        PRINT *,' 7. Do not print solution vector after each iterative r
     &efinement step'
      ELSE IF (BCTL(7).EQ.1) THEN
        PRINT *,' 7. Print solution vector after each iterative refineme
     &nt step'
      ELSE
        PRINT *,' 7. Invalid value!'
      ENDIF
```

```
C ---
      PRINT *,' '
      END
C ------------------------------------------------------------------------
C ---
C --- BCTL1: Get reporting option
C ---
      SUBROUTINE BCTL1 (CTL)
      IMPLICIT REAL*8(A-H,O-Z)
      N=-1
      DO WHILE (N.LT.0 .OR. N.GT.3)
        PRINT *,'BCTL(1)=',CTL,' : Reporting'
        PRINT *,' 0 = Display only severe errors'
        PRINT *,' 1 = Display severe errors and warnings'
        PRINT *,' 2 = Display severe errors, warnings, and progress mess
     &ages'
        PRINT *,' 3 = Display all messages plus statistics'
        PRINT *,'Enter value:'
        READ *,N
      ENDDO
      CTL=N
      END
C ------------------------------------------------------------------------
C ---
C --- BCTL2: Print solution vector?
C ---
      SUBROUTINE BCTL2 (CTL)
      IMPLICIT REAL*8(A-H,O-Z)
      N=-1
      DO WHILE (N.LT.0 .OR. N.GT.1)
        PRINT *,'BCTL(2)=',CTL,' : Print solution vector'
        PRINT *,' 0 = No'
        PRINT *,' 1 = Yes'
        PRINT *,'Enter value:'
        READ *,N
      ENDDO
      CTL=N
      END
C ------------------------------------------------------------------------
C ---
C --- BCTL3: Scale matrix before reducing or factoring it?
C ---
      SUBROUTINE BCTL3 (CTL)
      IMPLICIT REAL*8(A-H,O-Z)
      N=-1
      DO WHILE (N.LT.0 .OR. N.GT.1)
        PRINT *,'BCTL(3)=',CTL,' : Scale matrix before reducing or facto
     &ring it'
        PRINT *,' 0 = No'
        PRINT *,' 1 = Yes'
        PRINT *,'Enter value:'
        READ *,N
      ENDDO
      CTL=N
      END
C ------------------------------------------------------------------------
C ---
C --- BCTL4: Matrix permutations
C ---
      SUBROUTINE BCTL4 (CTL)
      IMPLICIT REAL*8(A-H,O-Z)
      N=-1
      DO WHILE (N.LT.0 .OR. N.GT.1)
```

```
        PRINT *,'BCTL(4)=',CTL,' : Matrix permutations'
        PRINT *,' 0 = Do not perform any permutations'
        PRINT *,' 1 = Partial pivoting by interchanging rows'
        PRINT *,'Enter value:'
        READ *,N
      ENDDO
      CTL=N
      END
C -----------------------------------------------------------------------
C ---
C --- BCTL5: Partial pivoting threshold
C ---
      SUBROUTINE BCTL5 (CTL)
      IMPLICIT REAL*8(A-H,O-Z)
      T=-1.D0
      DO WHILE (T.LT.0.D0 .OR. T.GT.1.D0)
        PRINT *,'BCTL(5)=',CTL,' : Partial pivoting threshold'
        PRINT *,' Must be between 0.D0 and 1.D0 inclusive.'
        PRINT *,'Enter value:'
        READ *,T
      ENDDO
      CTL=T
      END
C -----------------------------------------------------------------------
C ---
C --- BCTL6: Iterative refinement
C ---
      SUBROUTINE BCTL6 (CTL)
      IMPLICIT REAL*8(A-H,O-Z)
      PRINT *,'BCTL(6)=',CTL,' : Perform iterative refinement (LU soluti
     &ons only)'
      PRINT *,'  0 = No iterative refinement'
      PRINT *,' >0 = Number of refinement steps.  Stop early if no furth
     &re improvement'
      PRINT *,' <0 = -1 * exact number of refinement steps'
      PRINT *,'Enter value:'
      READ *,N
      CTL=N
      print *,'ctl=',ctl
      END
C -----------------------------------------------------------------------
C ---
C --- BCTL7: Print iterative refinement solution vector
C ---
      SUBROUTINE BCTL7 (CTL)
      IMPLICIT REAL*8(A-H,O-Z)
      N=-1
      DO WHILE (N.LT.0 .OR. N.GT.1)
        PRINT *,'BCTL(7)=',CTL,' : Print iterative refinement solution v
     &ector'
        PRINT *,' 0 = No'
        PRINT *,' 1 = Yes'
        PRINT *,'Enter value:'
        READ *,N
      ENDDO
      CTL=N
      END
C -----------------------------------------------------------------------
C ---
C --- BVALID: Validate BCTL settings
C ---
C --- This routine insures that invalid values do not enter BCTL() through
C --- the BINIT routine.
```

```
C ---
      SUBROUTINE BVALID (DONE,BCTL)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      LOGICAL DONE
      DONE=.TRUE.
      IF (BCTL(1).NE.0 .AND. BCTL(1).NE.1 .AND. BCTL(1).NE.2 .AND.
     &    BCTL(1).NE.3) DONE=.FALSE.
      IF (BCTL(2).NE.0 .AND. BCTL(2).NE.1) DONE=.FALSE.
      IF (BCTL(3).NE.0 .AND. BCTL(3).NE.1) DONE=.FALSE.
      IF (BCTL(4).NE.0 .AND. BCTL(4).NE.1) DONE=.FALSE.
      IF (BCTL(5).LT.0.D0 .OR. BCTL(5).GT.1.D0) DONE=.FALSE.
      IF (BCTL(7).NE.0 .AND. BCTL(7).NE.1) DONE=.FALSE.
      END
```

Appendix 7. Banded LU Factorization Routines

A7.1. blu.f

```
C-----------------------------------------------------------------------------
C   blu.f
C
C   Load and solve the set of linear equations Ax=b using banded LU
C   factorization and iterative refinement.
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------------
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),B(MAXROW)
      DIMENSION A1(MAXBW,MAXROW),NCOLL1(MAXROW),NCOLU1(MAXROW),
     &  B1(MAXROW)
      DIMENSION X(MAXROW),DX(MAXROW),R(MAXROW)
      SAVE A,NCOLL,NCOLU,B,A1,NCOLL1,NCOLU1,B1,X,DX,R
C ---
C --- Initialization
C ---
      PRINT *,'Banded LU Factorization with Iterative Refinement'
      PRINT *,'Rodney Jacobs, University of Maine, 2005'
      PRINT *,' '
      CALL BINIT (BCTL,BINFO)
      CALL BUSER (BCTL,BINFO)
C ---
C --- Read and scale data and load arrays.
C ---
      CALL BLOAD (A,NCOLL,NCOLU,B,BCTL,BINFO)
      CALL BSCALE (A,NCOLL,NCOLU,B,BCTL,BINFO)
      CALL BCOPY (A,NCOLL,NCOLU,B,A1,NCOLL1,NCOLU1,B1,BCTL,BINFO)
C ---
C --- Factor A to LU in place
C ---
      CALL BLUFAC (A,NCOLL,NCOLU,BCTL,BINFO)
C ---
C --- Solve LUx=b for x
C ---
      CALL BLUSOLVE (A,NCOLL,NCOLU,X,B,BCTL,BINFO)
C ---
C --- Report statistics
C ---
      CALL BLUSTATS (A,BCTL,BINFO)
C ---
C --- Compute error measures and residual vector R
C ---
      CALL BERROR (A1,NCOLL1,NCOLU1,X,B1,R,BCTL,BINFO)
C ---
C --- Iterative refinement
C ---
      CALL BREFINE (A,NCOLL,NCOLU,A1,NCOLL1,NCOLU1,X,B1,R,.TRUE.,
     &  BCTL,BINFO)
      END
```

A7.2. blufac.f

```
C-----------------------------------------------------------------------
C   blufac.f
C
C   Perform LU factorization of a banded matrix A without partial pivoting.
C
C   Input:
C     A             Matrix to be factored in banded storage format
C     NCOLL   Column index of leftmost non-zero term in row of array A
C     NCOLU   Column index of rightmost non-zero term in row of array A
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     Array A contains L and U factors.  Diagonal elements of L implicitly
C        have value = 1.
C     NCOLU may be updated
C     Set FLOP count and execution time in BINFO
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE BLUFAC (A,NCOLL,NCOLU,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW)
      DIMENSION ATEMP(MAXBW)
      REAL DTIME,TA(2)
C ---
C --- Initialization
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLUFAC: Factor A to LU form'
      IF (BCTL(4).EQ.1) THEN
        PRINT *,'BLUFAC: Partial pivoting is not compatible with LU fact
     &orization'
        STOP
      ENDIF
      NROW=BINFO(2)
      NBWL=BINFO(3)
      NBWL1=NBWL+1
      FLOP=0.D0
      TIME=DTIME(TA)
      DO NR=1,NROW
C --- Compute upper limit of rows affected
        K=NR+NBWL
        IF (K.GT.NROW) K=NROW
C --- Check for zero pivot
        PIVOT=A(NBWL1,NR)
        IF (PIVOT.EQ.0.D0) THEN
          PRINT *,'BLUFAC: Pivot = 0.D0'
          STOP
        ENDIF
C --- Factorization step
        DO MR=NR+1,K
          XMULT=A(NBWL1-MR+NR,MR)
          IF (XMULT.NE.0.D0) THEN
            XMULT=XMULT/PIVOT
            A(NBWL1-MR+NR,MR)=XMULT
            MC=0
            DO NC=NBWL1+1,NCOLU(NR)
              MC=NC-MR+NR
              A(MC,MR)=A(MC,MR)-XMULT*A(NC,NR)
```

```
        ENDDO
        IF (NCOLU(MR).LT.MC) NCOLU(MR)=MC
        NCNT=NCOLU(NR)-NBWL1
        IF (NCNT.LT.0) NCNT=0
        FLOP=FLOP+2*NCNT+1
      ENDIF
    ENDDO
  ENDDO
  BINFO(9)=FLOP
  BINFO(10)=DTIME(TA)
  END
```

A7.3. brefine.f

```
C-----------------------------------------------------------------------------
C   brefine.f
C
C   Banded iterative refinement.  BLUSOLVE must be called before calling this
C   routine.
C
C   Input:
C     A       LU in banded storage format
C     NCOLL   Column index of leftmost non-zero term in row of array A
C     NCOLU   Column index of rightmost non-zero term in row of array A
C     A1      Matrix of Ax=b in banded storage format
C     NCOLL1  Column index of leftmost non-zero term in row of array A1
C     NCOLU1  Column index of rightmost non-zero term in row of array A1
C     X       Solution vector of Ax=b
C     B1      Right hand side of Ax=b
C     R       Residual vector r=Ax-b
C     ERRSET  .TRUE. => Residual vector R computed by calling BERROR
C                 before calling this routine.  If .FALSE., BERROR will
C                 be called by this routine to compute R
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     X updated
C     R updated
C     BINFO updated.  See bandparam.h
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------------
      SUBROUTINE BREFINE (A,NCOLL,NCOLU,A1,NCOLL1,NCOLU1,X,B1,R,
     &  ERRSET,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),
     &  A1(MAXBW,MAXROW),NCOLL1(MAXROW),NCOLU1(MAXROW),X(MAXROW),
     &  B1(MAXROW),R(MAXROW)
      DIMENSION DX(MAXROW),X1(MAXROW)
      LOGICAL ERRSET
C ---
C --- Check to see if iterative refinement is to be performed
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BREFINE: Iterative refinement'
      BINFO(22)=0
      IF (BCTL(6).EQ.0) THEN
        IF (BCTL(1).GE.1)
     &    PRINT *,'BREFINE: BCTL(6)=0: Iterative refinement disabled'
        RETURN
      ENDIF
C ---
C --- Compute residual vector and error measures if not previously done
C ---
      IF (.NOT. ERRSET)
     &  CALL BERROR (A1,NCOLL1,NCOLU1,X,B1,R,BCTL,BINFO)
C ---
C --- Iterative refinement loop
C ---   1. Solve LU(dx)=r for dx vector where is r is residual vector
C ---   2. Compute new solution vector x1=x-dx
C ---   3. Compute new error measures and residual vector
C ---   4. If the solution improved, set x=x1 and repeat loop
C ---
```

```
      BCTL2=BCTL(2)
      BCTL(2)=0
      NROW=BINFO(2)
      BERR=BINFO(16)
      DO N=1,ABS(BCTL(6))
        CALL BLUSOLVE (A,NCOLL,NCOLU,DX,R,BCTL,BINFO)
        DO I=1,NROW
          X1(I)=X(I)-DX(I)
        ENDDO
        CALL BERROR (A1,NCOLL1,NCOLU1,X1,B1,R,BCTL,BINFO)
        IF (BCTL(6).GT.0 .AND. BINFO(16).GE.BERR) THEN
          IF (BCTL(1).GE.2)
     &      PRINT *,'BREFINE: No further improvement'
          GOTO 10
        ENDIF
        BERR=BINFO(16)
        DO I=1,NROW
          X(I)=X1(I)
        ENDDO
        BINFO(22)=BINFO(22)+1
      ENDDO
10    BCTL(2)=BCTL2
      IF (BCTL(1).GE.2) THEN
        N=BINFO(22)
        PRINT *,'BREFINE: ',N,' refinement step(s) performed'
      ENDIF
C ---
C --- Print x
C ---
      IF (BCTL(7).EQ.1) THEN
        DO I=1,NROW
          PRINT *,'BREFINE: X(',I,') =',X(I)
        ENDDO
      ENDIF
      END
```

A7.4. blusolve.f

```
C-----------------------------------------------------------------------------
C   blusolve.f
C
C   Solve LUx=b using forward and backward substitution where LU is in
C   banded storage format.
C
C   Input:
C     A       L and U matrices in banded storage format.  Diagonal
C               elements of L have implicit value = 1.
C     B       Right hand side of LUx=b
C     NCOLL   Column index of leftmost non-zero term in row of array LU
C     NCOLU   Column index of rightmost non-zero term in row of array LU
C     NPERM   Row permutation vector. NPERM(I) is the row of matrix A
C               represented by row I of L and U.
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     Set X to solution vector
C     Set FLOP count and execution time in BINFO
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------------
      SUBROUTINE BLUSOLVE (A,NCOLL,NCOLU,X,B,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW),NCOLL(MAXROW),NCOLU(MAXROW),X(MAXROW),
     &  B(MAXROW),Y(MAXROW)
      REAL DTIME,TA(2)
C ---
C --- Solve Ly=b
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLUSOLVE: Solve Ly=b using forward substitution'
      NROW=BINFO(2)
      NBWL=BINFO(3)
      NBWL1=NBWL+1
      FLOP=0.D0
      TIME=DTIME(TA)
      DO NR=1,NROW
        XSUM=B(NR)
        K=NR-(NBWL1-NCOLL(NR))
        DO J=K,NR-1
          NC=J-NR+NBWL1
          XSUM=XSUM-A(NC,NR)*Y(J)
        ENDDO
        Y(NR)=XSUM
        NCNT=NR-K
        IF (NCNT.LT.0) NCNT=0
        FLOP=FLOP+2*NCNT
      ENDDO
C ---
C --- Solve Ux=y
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLUSOLVE: Solve Ux=y using backward substitution'
      DO NR=NROW,1,-1
        XSUM=Y(NR)
        K=NR+(NCOLU(NR)-NBWL1)
        DO J=NR+1,K
          NC=J-NR+NBWL1
```

```
          XSUM=XSUM-A(NC,NR)*X(J)
        ENDDO
        X(NR)=XSUM/A(NBWL1,NR)
        NCNT=K-NR
        IF (NCNT.LT.0) NCNT=0
        FLOP=FLOP+2*(NCNT)+1
      ENDDO
      BINFO(11)=FLOP
      BINFO(12)=DTIME(TA)
C ---
C --- Print x
C ---
      IF (BCTL(2).EQ.1) THEN
        DO NR=1,NROW
          PRINT *,'X(',NR,') =',X(NR)
        ENDDO
      ENDIF
      END
```

A7.5. blustats.f

```
C-----------------------------------------------------------------------
C   blustats.f
C
C   Display LU factorization and solve statistics
C
C   Input:
C     A       LU in banded storage format
C     BCTL    See bandparam.h
C     BINFO   See bandparam.h
C
C   Side effects:
C     Set L and U non-zero entry counts in BINFO
C
C   Rodney Jacobs, University of Maine, 2005
C-----------------------------------------------------------------------
      SUBROUTINE BLUSTATS (A,BCTL,BINFO)
      IMPLICIT REAL*8(A-H,O-Z)
      INCLUDE "bandparam.h"
      DIMENSION A(MAXBW,MAXROW)
C ---
C --- Count non-zero entries in L, U, and diagonal
C ---
      IF (BCTL(1).GE.2)
     &  PRINT *,'BLUSTATS: Count non-zero entries in L and U'
      NROW=BINFO(2)
      NBWL=BINFO(3)
      NBWL1=NBWL+1
      NBWU=BINFO(4)
      NZL=0
      NZD=0
      NZU=0
      DO I=1,NROW
        DO J=1,NBWL
          IF (A(J,I).NE.0.D0) NZL=NZL+1
        ENDDO
        IF (A(NBWL1,I).NE.0.D0) NZD=NZD+1
        DO J=NBWL1+1,NBWL1+NBWU
          IF (A(J,I).NE.0.D0) NZU=NZU+1
        ENDDO
      ENDDO
      BINFO(13)=NZL
      BINFO(14)=NZD
      BINFO(15)=NZU
C ---
C --- Print statistics
C ---
      IF (BCTL(1).GE.3) THEN
        PRINT *,'LU Statistics'
        PRINT *,'----------------------------------------------'
        PRINT *,'Non-zeros in L w/diagonal..: ',BINFO(13)+BINFO(2)
        PRINT *,'Non-zeros in U w/diagonal..: ',BINFO(14)+BINFO(15)
        PRINT *,'Non-zeros in L+U...........: ',BINFO(13)+BINFO(14)+
     &                                          BINFO(15)
        PRINT *,'LU Factorization Time (sec): ',BINFO(10)
        PRINT *,'Substitution Time (sec)....: ',BINFO(12)
        PRINT *,'Total Solve Time (sec).....: ',BINFO(10)+BINFO(12)
        PRINT *,'LU Factorization FLOPS.....: ',BINFO(9)
        PRINT *,'Substitution FLOPS.........: ',BINFO(11)
        PRINT *,'Total Solve FLOPS..........: ',BINFO(9)+BINFO(11)
      ENDIF
      END
```

BIOGRAPHY OF THE AUTHOR

Rodney Jacobs was born in Bangor, Maine on October 1, 1954. He was raised in Bucksport, Maine and graduated from Bucksport High School in 1972. He attended the Massachusetts Institute of Technology and graduated in 1976 with a Bachelor of Science degree in Electrical Engineering. After working in Woburn, Massachusetts for a short while as a junior microwave design engineer, he returned to Maine and was a computer programmer for Data Systems of Maine until 1978. He moved on to work at Great Northern Paper Company as a programmer/analyst until 1980. After leaving Great Northern Paper Company he worked for 16 years as an independent software developer writing campaign and fund accounting software for United Ways that has been used by over 200 organizations throughout the country, as well as business software systems for automobile dealers, banking, retail and wholesale distribution, agricultural nurseries, and others. For the past ten years he has worked for N. H. Bragg and Sons in Bangor, Maine as a software developer and information systems manager. Rodney lives in Bangor with his wife Susan and their daughter Rebecca.

Learning has been a life long passion for Rodney. His graduate work at The University of Maine has been a source of professional development and personal reward. Rodney is a candidate for the Master of Science degree in Computer Science from The University of Maine in December, 2005.