Matrix Algorithms

Timothy Vismor
January 30, 2015

Abstract

This document examines various aspects of matrix and linear algebra that are relevant to the analysis of large scale networks. Particular emphasis is placed on computational aspects of the topics of interest.

Contents

List of Tables

List of Figures

List of Algorithms

1 Matrix Nomenclature

Since any finite dimensional linear operator can be represented as a matrix, matrix algebra and linear algebra are two sides of the same coin. Properties of linear systems are gleaned from either discipline. The following sections draw on both of these perspectives to examine the basic concepts, numerical techniques, and practical constraints of computational linear algebra.

Assuming the symbols x_i represent variables and the symbols a_{ij} and $b_i$are complex constants, the following is a system of m linear equations in n unknowns.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{1}$$

This system of equations is expressed in matrix notation as

$$\mathbf{Ax} = \mathbf{b} \tag{2}$$

where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{pmatrix} \tag{3}$$

A rectangular array of coefficients such as a $\mathbf{A}$ is referred to as a matrix. The matrix $\mathbf{A}$ has m rows and n columns. As such, it is called an $m \times n$ matrix. A square matrix has an equal number of rows and columns, e.g. an $n \times n$ matrix. A vector is a matrix with just one row or just one column. A $1 \times n$ matrix is a row vector. An $m \times 1$ matrix, such as $\mathbf{x}$ or $\mathbf{b}$ in Equation 2, is called a column vector.

The elements of a matrix a_{ii} whose row and column index are equal are referred to as its diagonal. The elements of a matrix above the diagonal (a_{ij}, where $i < j$) are its superdiagonal entries. The elements of a matrix below the diagonal (a_{ij}, where $i > j$) are its subdiagonal entries. A matrix whose subdiagonal entries are zero is called upper triangular. An upper triangular matrix with ones along the diagonal is called unit upper triangular. The following 3×3 matrix is unit upper triangular.

$$\begin{pmatrix} 1 & a_{12} & a_{13} \\ 0 & 1 & a_{23} \\ 0 & 0 & 1 \end{pmatrix}$$

Similarly, a matrix whose superdiagonal entries are zero is called lower triangular. A lower triangular matrix with ones along the diagonal is called unit lower triangular. The

following 3 × 3 matrix is lower triangular.

$$\begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

A matrix whose superdiagonal and subdiagonal entries are zero is a diagonal matrix, e.g.

$$\begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}$$

A square matrix whose subdiagonal elements are the mirror image of its superdiagonal elements is referred to as a symmetric matrix. More formally, a symmetric matrix $\mathbf{A}$ has the property $a_{ij} = a_{ji}$. A trivial example of a symmetric matrix is a diagonal matrix. The general case of a 3 × 3 symmetric matrix follows.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}$$

A matrix whose elements are all zero is called the zero matrix or the null matrix.

2 Matrix Algebra

The set of square matrices of dimension n form an algebraic entity known as a ring. By definition, a ring consists of a set R and two operators (addition + and multiplication ×) such that

- R is an Abelian group with respect to addition.
- R is a semigroup with respect to multiplication.
- R is left distributive, i.e. a × (b + c) = (a × b) + (a × c).
- R is right distributive, i.e.(b + c) × a = (b × a) + (c × a).

An Abelian group consists of a set G and a binary operator such that

- G is associative with respect to the operator.
- G has an identity element with respect to the operator.
- Each element of G has an inverse with respect to the operator.
- G is commutative with respect to the operator.

A semigroup consists of a set G and a binary operator such that G is associative with respect to the operator.

For non-square matrices, even these limited properties are not generally true. The following sections examine the algebraic properties of matrices in further detail.

2.1 Matrix Equality

Two $m \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$ are equal if their corresponding elements are equal.

$$\mathbf{A} = \mathbf{B} \tag{4}$$

implies

$$a_{ij} = b_{ij}, \text{where } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \tag{5}$$

The notion of matrix equality is undefined unless the operands have the same dimensions.

2.2 Matrix Transposition

The transpose of an $m \times n$ matrix $\mathbf{A}$ is an $n \times m$ matrix denoted by $\mathbf{A^T}$. The columns of $\mathbf{A^T}$ are the rows of $\mathbf{A}$ and the rows of $\mathbf{A^T}$ are the columns of $\mathbf{A}$.

$$a_{ij}^T = a_{ji}, \text{where } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \tag{6}$$

A symmetric matrix is its own transpose, i.e. if $\mathbf{A}$ is symmetric

$$\mathbf{A} = \mathbf{A^T}$$

The transpose of the 2×3 matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

is the 3×2 matrix

$$\begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \end{pmatrix}$$

2.3 Scalar Multiplication

The product of an $m \times n$ matrix $\mathbf{A}$ and a scalar α is an $m \times n$ matrix whose elements are the arithmetic products of α and the elements of $\mathbf{A}$.

$$\mathbf{C} = \alpha \cdot \mathbf{A} \tag{7}$$

implies

$$c_{ij} = \alpha \cdot a_{ij}, \text{where } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \tag{8}$$

2.4 Matrix Addition

The sum of $m \times n$ matrices $\mathbf{A}$ and $\mathbf{B}$ is an $m \times n$ matrix $\mathbf{C}$ which is the element by element sum of the addends.

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \tag{9}$$

implies

$$c_{ij} = a_{ij} + b_{ij}, \text{where } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \tag{10}$$

Matrix addition is undefined unless the addends have the same dimensions. Matrix addition is commutative.

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$$

Matrix addition is also associative.

$$(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C})$$

The additive identity is the zero matrix. The additive inverse of matrix $\mathbf{A}$ is denoted by -$\mathbf{A}$ and consists of the element by element negation of a $\mathbf{A}$, i.e. it's the matrix formed when a $\mathbf{A}$ is multiplied by the scalar -1.

$$-\mathbf{A} = -1 \cdot \mathbf{A} \tag{11}$$

2.5 Matrix Multiplication

The product of an $m \times p$ matrix $\mathbf{A}$ and a $p \times n$ matrix $\mathbf{B}$ is an $m \times n$ matrix $\mathbf{C}$ where each element c_{ij} is the dot product of row i of $\mathbf{A}$ and column j of $\mathbf{B}$.

$$\mathbf{C} = \mathbf{AB} \tag{12}$$

implies

$$c_{ij} = \sum_{k=1}^{p} (a_{ik} + b_{kj}), \text{where } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \tag{13}$$

The product of matrices $\mathbf{A}$ and $\mathbf{B}$ is undefined unless the number of rows in $\mathbf{A}$ is equal to the number of columns in $\mathbf{B}$. In this case, the matrices are conformable for multiplication.

In general, matrix multiplication is not commutative.

$$\mathbf{AB} \neq \mathbf{BA}$$

As a consequence, the following terminology is sometimes used. Considering the matrix product

$$\mathbf{AB}$$

The left multiplicand $\mathbf{A}$ is said to premultiply the matrix $\mathbf{B}$. The right multiplicand $\mathbf{B}$ is said to postmultiply the matrix $\mathbf{A}$.

Matrix multiplication distributes over matrix addition

$$\mathbf{A}(\mathbf{B}+\mathbf{C}) = (\mathbf{AB}) + (\mathbf{AC})$$

and

$$(\mathbf{B}+\mathbf{C})\mathbf{A} = (\mathbf{BA}) + (\mathbf{CA})$$

if $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are conformable for the indicated operations. With the same caveat, matrix multiplication is associative.

$$\mathbf{A}\,(\mathbf{BC}) = (\mathbf{AB})\,\mathbf{C}$$

The transpose of a matrix product is the product of the factors in reverse order, i.e.

$$(\mathbf{ABC})^{\mathbf{T}} = \mathbf{C}^{\mathbf{T}}\mathbf{B}^{\mathbf{T}}\mathbf{A}^{\mathbf{T}} \tag{14}$$

The set of square matrices has a multiplicative identity which is denoted by $\mathbf{I}$. The identity is a diagonal matrix with ones along the diagonal

$$a_{ij} = \begin{cases} 1 \text{ where } i = j \\ 0 \text{ where } i \neq j \end{cases} \tag{15}$$

The 3×3 multiplicative identity is

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2.6 Inverse of a Matrix

If $\mathbf{A}$ and $\mathbf{B}$ are square $n \times n$ matrices such that

$$\mathbf{AB} = \mathbf{I} \tag{16}$$

then $\mathbf{B}$ is a right inverse of $\mathbf{A}$. Similarly, if $\mathbf{C}$ is an $n \times n$ matrix such that

$$\mathbf{CA} = \mathbf{I} \tag{17}$$

then **C** is a left inverse of **A**. When both Equation 16 and Equation 17 hold

$$\mathbf{AB} = \mathbf{CA} = \mathbf{I} \tag{18}$$

then **B** = **C** and **B** is the two-sided inverse of **A**.

The two-sided inverse of **A** will be referred to as its multiplicative inverse or simply its inverse. If the inverse of **A** exists, it is unique and denoted by $\mathbf{A}^{-1}$. $\mathbf{A}^{-1}$ exists if and only if **A** is square and nonsingular. A square $n \times n$ matrix is singular when its rank is less than n, i.e. two or more of its columns (or rows) are linearly dependent. The rank of a matrix is examined more closely in Section 2.7 of this document.

A few additional facts about inverses. If **A** is invertible, so is $\mathbf{A}^{-1}$ and

$$\left(\mathbf{A}^{-1}\right)^{-1} = \mathbf{A} \tag{19}$$

If **A** and **B** are invertible, so is **AB** and

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1} \tag{20}$$

Extending the previous example

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1} \tag{21}$$

If **A** is invertible, then

$$\left(\mathbf{A}^{-1}\right)^{\mathrm{T}} = \left(\mathbf{A}^{\mathrm{T}}\right)^{-1} \tag{22}$$

The conditional or generalized inverse which may be defined for any matrix is beyond the scope of the current discussion.

2.7 Rank of a Matrix

The rank of an $n \times n$ matrix **A** is the maximum number of linearly independent columns in **A**. Column vectors of of **A**, denoted $\mathbf{a}^{\mathbf{i}}$, are linearly independent if the only set of scalars α_i such that

$$\alpha_1\mathbf{a}^{\mathbf{1}} + \alpha_2\mathbf{a}^{\mathbf{2}} + \ldots + \alpha_n\mathbf{a}^{\mathbf{n}} = \mathbf{0} \tag{23}$$

is the set

$$\alpha_1 = \alpha_2 = \ldots = \alpha_n = 0$$

For a more concrete example, consider the following matrix.

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 1 & 2 & 3 & 4 \\ 2 & 0 & 2 & 0 \end{pmatrix}$$

The rank of $\mathbf{A}$ is two, since its third and fourth columns are linear combinations of its first two columns, i.e.

$$\mathbf{a}^3 = \mathbf{a}^1 + \mathbf{a}^2$$
$$\mathbf{a}^4 = 2\mathbf{a}^2$$

If $\mathbf{A}$ is an $n \times n$ matrix, it can be shown

$$\text{rank}\,(\mathbf{A}) \leq \min(m, n) \tag{24}$$

Furthermore,

$$\text{rank}\,(\mathbf{AB}) \leq \min(\text{rank}\,(\mathbf{A}), \text{rank}\,(\mathbf{B})) \tag{25}$$

and

$$\text{rank}\left(\mathbf{AA}^{\mathbf{T}}\right) = \text{rank}\left(\mathbf{A}^{\mathbf{T}}\mathbf{A}\right) = \text{rank}\,(\mathbf{A}) \tag{26}$$

2.8 Similarity Transformations

If $\mathbf{A}$ and $\mathbf{B}$ are $n \times n$ matrices, $\mathbf{A}$ is similar to $\mathbf{B}$ if there exists an invertible matrix $\mathbf{P}$ such that

$$\mathbf{B} = \mathbf{PAP}^{-1} \tag{27}$$

Every matrix is similar to itself with $\mathbf{P} = \mathbf{I}$. The only similarity transformation that holds for the identity matrix or the zero matrix is this trivial one.

Similarity is a symmetric relation. If $\mathbf{A} \sim \mathbf{B}$, then $\mathbf{B} \sim \mathbf{A}$. Therefore, premultiplying Equation 27 by $\mathbf{P}^{-1}$ and postmultiplying it by $\mathbf{P}$ yields

$$\mathbf{A} = \mathbf{P}^{-1}\mathbf{BP} \tag{28}$$

Similarity is also a transitive relation. If $\mathbf{A} \sim \mathbf{B}$ and $\mathbf{B} \sim \mathbf{C}$, then $\mathbf{A} \sim \mathbf{C}$.

Since similarity is reflexive, symmetric, and transitive, it is an equivalence relation. A common example of a similarity transformation in linear algebra is changing the basis of a vector space.

2.9 Partitioning a Matrix

Matrices may be divided into subsections for computational purposes. Consider the $n \times n$ matrix $\mathbf{A}$ which is partitioned along the following lines.

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \tag{29}$$

If $k \times k$ matrix $\mathbf{A}_{11}$ and $p \times p$ matrix $\mathbf{A}_{22}$ are square matrices, then $\mathbf{A}_{12}$ has dimensions $k \times p$ and $\mathbf{A}_{21}$ has dimensions $p \times k$.

The transpose of $\mathbf{A}$ is

$$\mathbf{A}^{\mathrm{T}} = \begin{pmatrix} \mathbf{A}_{11}^{\mathrm{T}} & \mathbf{A}_{12}^{\mathrm{T}} \\ \mathbf{A}_{21}^{\mathrm{T}} & \mathbf{A}_{22}^{\mathrm{T}} \end{pmatrix} \tag{30}$$

If $\mathbf{A}$ is invertible, its inverse is

$$\mathbf{A}^{-1} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix} \tag{31}$$

where

$$\begin{aligned} \mathbf{B}_{11} &= (\mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21})^{-1} \\ \mathbf{B}_{12} &= -\mathbf{A}_{11}^{-1}\mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{B}_{21} &= -\mathbf{A}_{22}^{-1}\mathbf{A}_{21}\mathbf{B}_{11} \\ \mathbf{B}_{22} &= (\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})^{-1} \end{aligned} \tag{32}$$

Alternately,

$$\begin{aligned} \mathbf{B}_{12} &= -\mathbf{B}_{11}\mathbf{A}_{12}\mathbf{A}_{22}^{-1} \\ \mathbf{B}_{22} &= \mathbf{A}_{22}^{-1} - \mathbf{A}_{22}^{-1}\mathbf{A}_{21}\mathbf{B}_{12} \end{aligned} \tag{33}$$

The product of $\mathbf{A}$ and another $n \times n$ matrix $\mathbf{B}$ which is partitioned along the the same lines is an identically partitioned matrix $\mathbf{C}$ such that

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\ \mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\ \mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{aligned} \tag{34}$$

The current discussion has focused on the principal partition of a square matrix; however, all aspects of the discussion (except the inversion rules) are more general – provided the dimensions of the partitions are conformable for the indicated operations.

3 Linear Systems

Consider the $m \times n$ system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{35}$$

If $m = n$ and $\mathbf{A}$ is not singular, Equation 35 possesses a unique solution and is referred to as a fully determined system of equations. When $m > n$ (or $m = n$ and $\mathbf{A}$ is singular), Equation 35 is an underdetermined system of equations. Otherwise, $m < n$ and Equation 35 is overdetermined system of equations.

3.1 Solving Fully Determined Systems

A $m \times n$ system of linear equations where $m = n$ and $\mathbf{A}$ is not singular, possesses a unique solution. A solution for the unknown vector $\mathbf{x}$ is obtained by premultiplying Equation 35 by $\mathbf{A}^{-1}$, i.e.

$$\left(\mathbf{A}^{-1}\mathbf{A}\right)\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

or simply

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{36}$$

since $\mathbf{A}^{-1}\mathbf{A}$ is by definition the multiplicative identity.

The solution algorithm suggested by Equation 36 is

1. Invert matrix $\mathbf{A}$.
2. Premultiply the vector $\mathbf{b}$ by $\mathbf{A}^{-1}$.

This procedure is computationally inefficient and rarely used in practice. Most direct solutions to systems of linear equations are derived from a procedure known as Gaussian elimination, which is a formalization of the *ad hoc* techniques used to solve linear equations in high school algebra. The basic algorithm is

1. Transform the system of equations into a triangular form.
2. Solve the triangular set of equations through a series of variable substitutions.

A major drawback to this procedure is that both sides ($\mathbf{A}$ and $\mathbf{b}$) of Equation 35 are modified as the system is forced into triangular form. This requires repetition of the entire procedure if you want to solve Equation 35 with a new $\mathbf{b}$ vector. A technique known as LU decomposition overcomes this problem by systematically capturing the intermediate states of $\mathbf{A}$ as the transformation to triangular form progresses. This effectively decouples the operations on $\mathbf{A}$ from those on $\mathbf{b}$, permitting solutions to Equation 35 for many $\mathbf{b}$ vectors based on a single triangular factorization.

More specifically, LU decomposition produces a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ such that

$$\mathbf{LU} = \mathbf{A} \tag{37}$$

Substituting Equation 37 into Equation 35 yields

$$(\mathbf{LU})\,\mathbf{x} = \mathbf{b} \tag{38}$$

Associating the factors in Equation 38 yields

$$\mathbf{L}\,(\mathbf{Ux}) = \mathbf{b} \tag{39}$$

Recalling that efficient procedures exist for solving triangular systems (i.e. forward substitution for lower triangular systems and backward substitution for upper triangular

systems), Equation 39 suggests an algorithm for solving Equation 35. Define a vector $\mathbf{y}$ such that

$$\mathbf{y} = \mathbf{U}\mathbf{x} \tag{40}$$

Substituting Equation 40 into Equation 39 yields

$$\mathbf{L}\mathbf{y} = \mathbf{b} \tag{41}$$

Since $\mathbf{b}$ is known and $\mathbf{L}$ is lower triangular, Equation 41 can be solved for $\mathbf{y}$ by forward substitution. Once $\mathbf{y}$ is known, Equation 40 can be solved for $\mathbf{x}$ by back substitution.

In summary, the preferred algorithm for solving for a nonsingular $n \times n$ system of linear equations is

1. Compute an LU decomposition of $\mathbf{A}$.
2. Solve Equation 41 for $\mathbf{y}$ by forward substitution.
3. Solve Equation 40 for $\mathbf{x}$ by back substitution.

3.2 Solving Underdetermined Systems

A $m \times n$ system of linear equations where $m > n$ (or $m = n$ and $\mathbf{A}$ is singular) is underdetermined. There are fewer equations than there are unknowns. Underdetermined systems have q linearly independent families of solutions, where

$$q = n - r$$

and

$$r = \text{rank}\,(\mathbf{A})$$

The value q is referred to as the nullity of matrix $\mathbf{A}$. The q linearly dependent equations in $\mathbf{A}$ are the null space of $\mathbf{A}$.

"Solving" an underdetermined set of equations usually boils down to solving a fully determined $r \times r$ system (known as the range of $\mathbf{A}$) and adding this solution to any linear combination of the other q vectors of $\mathbf{A}$. A numerical procedure that solves the crux of this problem is known as singular value decomposition (or SVD). A singular value decomposition constructs a set of orthonormal bases for the null space and range of $\mathbf{A}$.

3.3 Solving Overdetermined Systems

A $m \times n$ system of linear equations with $m < n$ is overdetermined. There are more equations than there are unknowns. "Solving" this equation is the process of reducing the

system to an $m \times m$ problem then solving the reduced set of equations. A common technique for constructing a reduced set of equations is known as the least squares solution to the equations. The least squares equations are derived by premultiplying Equation 35 by $\mathbf{A}^{\mathbf{T}}$, i.e.

$$\left(\mathbf{A}^{\mathbf{T}}\mathbf{A}\right)\mathbf{x} = \mathbf{A}^{\mathbf{T}}\mathbf{b} \tag{42}$$

Often Equation 42 is referred to as the normal equations of the linear least squares problem. The least squares terminology refers to the fact that the solution to Equation 42 minimizes the sum of the squares of the differences between the left and right sides of Equation 35.

3.4 Computational Complexity of Linear Systems

As was mentioned in Section 3.1, the decomposition algorithm for solving linear equations is motivated by the computational inefficiency of matrix inversion. Inverting a dense matrix $\mathbf{A}$ requires

$$2n^3 + O\left(n^2\right)$$

floating point operations. Computing the LU decomposition of $\mathbf{A}$ requires

$$\frac{2}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

or

$$\frac{2}{3}n^3 + O\left(n^2\right)$$

floating point operations. Computing $\mathbf{x}$ from the factorization requires

$$2n^2 + n$$

or

$$2n^2 + O\left(n\right)$$

floating point operations (which is equivalent to computing the product $\mathbf{A}^{-1}\mathbf{b}$). Therefore, solving a linear system of equations by matrix inversion requires approximately three times the amount of work as a solution via LU decomposition.

When $\mathbf{A}$ is a sparse matrix, the computational discrepancy between the two methods becomes even more overwhelming. The reason is straightforward. In general,

- Inversion destroys the sparsity of $\mathbf{A}$, whereas
- LU decomposition preserves the sparsity of $\mathbf{A}$.

Much work can be avoided by taking advantage of the sparsity of a matrix and its triangular factors. Algorithms for solving sparse systems of equations are are described in detail in Section 8 of this document.

4 LU Decomposition

There are many algorithms for computing the LU decomposition of the matrix **A**. All algorithms derive a matrix **L** and a matrix **U** that satisfy Equation 37. Most algorithms also permit **L** and **U** to occupy the same amount of space as **A**. This implies that either **L** or **U** is computed as a unit triangular matrix so that explicit storage is not required for its diagonal (which is all ones).

There are two basic approaches to arriving at an LU decomposition:

- Simulate Gaussian elimination by using row operations to zero elements in **A** until an upper triangular matrix exists. Save the multipliers produced at each stage of the elimination procedure as **L**.
- Use the definition of matrix multiplication to solve Equation 37 directly for the elements of **L** and **U**.

Discussions of the subject by Fox (1964), Golub and Van Loan (1983), Duff, Erisman, and Reid (1986), and Press et al. (1988) are complementary in many respects. Taken as a group, these works provide a good sense of perspective concerning the problem.

4.1 Gaussian Elimination

Approaches to LU decomposition which systematically capture the intermediate results of Gaussian elimination often differ in the order in which **A** is forced into upper triangular form. The most common alternatives are to eliminate the subdiagonal parts of **A** either one row at a time or one column at a time. The calculations required to zero a complete row or a complete column are referred to as one stage of the elimination process.

The effects of the k^{th} stage of Gaussian elimination on the **A** matrix are summarized by the following equation.

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - \left(\frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \right) a_{ij}^{(k)}, \text{where } i, j > k \tag{43}$$

The notation $a_{ij}^{(k)}$ means the value of a_{ij} produced during the k^{th} stage of the elimination procedure. In Equation 43, the term $\frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$ (sometimes referred to as a multiplier) captures the crux of the elimination process. It describes the effect of eliminating element a_{ik} on the other entries in row i during the k^{th} stage of the elimination. In fact, these multipliers are the elements of the lower triangular matrix **L**, i.e.

$$l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \tag{44}$$

Algorithm 1: LU Decomposition

for $k = 1, \cdots, \min(m-1, n)$
  for $j = k+1, \cdots, n$
    $w_j = a_{kj}$
  for $i = k+1, \cdots, m$
    $\alpha = \dfrac{a_{ik}}{a_{kk}}$
    $a_{ik} = \alpha$
    for $j = k+1, \cdots, n$
      $a_{ij} = a_{ij} - \alpha w_j$

Algorithm 1 implements Equation 43 and Equation 44 and computes the LU decomposition of an $m \times n$ matrix $\mathbf{A}$. It is based on Algorithm 4.2–1 of Golub and Van Loan (1983).

The algorithm overwrites a_{ij} with l_{ij} when $i < j$. Otherwise, a_{ij} is overwritten by u_{ij}. The algorithm creates a matrix $\mathbf{U}$ that is upper triangular and a matrix $\mathbf{L}$ that is unit lower triangular. Note that a working vector $\mathbf{w}$ of length n is required by the algorithm.

4.2 Doolittle's LU Factorization

An LU decomposition of $\mathbf{A}$ may be obtained by applying the definition of matrix multiplication to the equation $\mathbf{A} = \mathbf{LU}$. If L is unit lower triangular and U is upper triangular, then

$$a_{ij} = \sum_{p=1}^{\min(i,j)} l_{ip} u_{pj}, \text{where } 1 \leq i, j \leq n \tag{45}$$

Rearranging the terms of Equation 45 yields

$$l_{ij} = \frac{a_{ij} - \sum_{p=1}^{j-1} l_{ip} u_{pj}}{u_{jj}}, \text{where } i > j \tag{46}$$

and

$$u_{ij} = a_{ij} - \sum_{p=1}^{i-1} l_{ip} u_{pj}, \text{where } 1 \leq j \tag{47}$$

Jointly Equation 46 and Equation 47 are referred to as Doolittle's method of computing the LU decomposition of A. Algorithm 2 implements Doolittle's method. Calculations

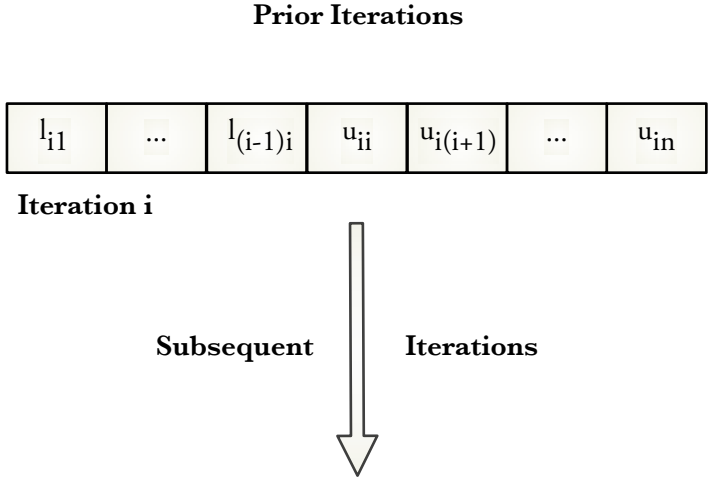
Algorithm 2: Doolittle's LU Decompostion

for $i = 1, \cdots, n$
  for $j = 1, \cdots, i-1$
    $\alpha = a_{ij}$
    for $p = 1, \cdots, j-1$
      $\alpha = \alpha - a_{ip}a_{pj}$
    $a_{ij} = \dfrac{\alpha}{a_{jj}}$
  for $j = i, \cdots, n$
    $\alpha = a_{ij}$
    for $p = 1, \cdots, i-1$
      $\alpha = \alpha - a_{ip}a_{pj}$
    $a_{ij} = \alpha$

are sequenced to compute one row of **L** followed by the corresponding row of **U** until **A** is exhausted. You will observe that the procedure overwrites the subdiagonal portions of **A** with **L** and the upper triangular portions of **A** with **U**.

Figure 1 depicts the computational sequence associated with Doolittle's method.

Figure 1: Computational Sequence of Doolittle's Method



Two loops in the Doolittle algorithm are of the form

$$\alpha = \alpha - a_{ip}a_{pj} \tag{48}$$

These loops determine an element of the factorization l_{ij} or u_{ij} by computing the dot product of a partial column vector in **U** and partial row vector in **L**. As such, the loops

perform an inner product accumulation. These computations have a numerical advantage over the gradual accumulation of l_{ij} or u_{ij} during each stage of Gaussian elimination (sometimes referred to as a partial sums accumulation). This advantage is based on the fact the product of two single precision floating point numbers is always computed with double precision arithmetic (at least in the C programming language). Because of this, the product $a_{ip}a_{pj}$ suffers no loss of precision. If the product is accumulated in a double precision variable α, there is no loss of precision during the entire inner product calculation. Therefore, one double precision variable can preserve the numerical integrity of the inner product.

Recalling the partial sum accumulation loop of the elimination-based procedure,

$$a_{ij} = a_{ij} - \alpha w_j$$

You will observe that truncation to single precision must occur each time a_{ij} is updated unless both **A** and **w** are stored in double precision arrays.

The derivation of Equation 46 and Equation 47 is discussed more fully in Conte and Boor (1972).

4.3 Crout's LU Factorization

An equivalent LU decomposition of **A** = **LU** may be obtained by assuming that **L** is lower triangular and **U** is unit upper triangular. This factorization scheme is referred to as Crout's method. The defining equations for Crout's method are

$$l_{ij} = a_{ij} - \sum_{p=1}^{i-1} l_{ip}u_{pj}, \text{where } i \geq j \tag{49}$$

and

$$u_{ij} = \frac{a_{ij} - \sum_{p=1}^{i-1} l_{ip}u_{pj}}{l_{ii}}, \text{where } i < j \tag{50}$$

Algorithm 3 implements Crout's method. Calculations are sequenced to compute one column of **L** followed by the corresponding row of **U** until **A** is exhausted.

Figure 2 depicts the computational sequence associated with Crout's method.

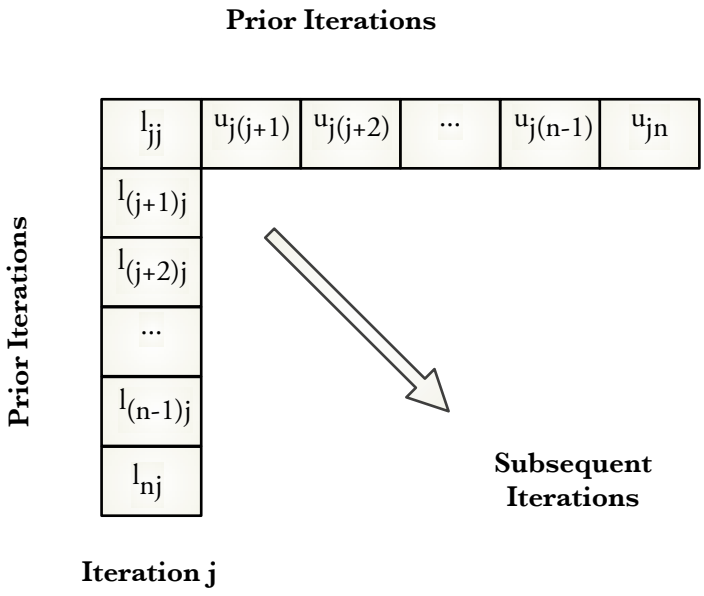
You should observe that Crout's method, like Doolittle's, exhibits inner product accumulation.

A good comparison of the various compact factorization schemes is found in Duff, Erisman, and Reid (1986).

Algorithm 3: Crout's LU Decomposition

for $j = 1, \cdots, n$
  for $i = j, \cdots, n$
    $\alpha = a_{ij}$
    for $p = 1, \cdots, j-1$
      $\alpha = \alpha - a_{ip}a_{pj}$
    $a_{ij} = \alpha$
  for $j = j+1, \cdots, n$
    $\alpha = a_{ij}$
    for $p = 1, \cdots, i-1$
      $\alpha = \alpha - a_{jp}a_{pi}$
    $a_{ji} = \frac{\alpha}{a_{jj}}$

Figure 2: Computational Sequence of Crout's Method



4.4 LDU Factorization

Some factorization algorithms, referred to as LDU decompositions, derive three matrices **L**, **D**, and **U** from **A** such that

$$\mathbf{LDU} = \mathbf{A} \tag{51}$$

where **L** is unit upper triangular, **D** is diagonal, and **U** is unit lower triangular. It should be obvious that the storage requirements of LDU decompositions and LU decompositions are the same.

A procedure proposed by Tinnney and Walker (1967) provides a concrete example of an LDU decomposition that is based on Gaussian elimination. One row of the subdiagonal portion of **A** is eliminated at each stage of the computation. Tinney refers to the LDU decomposition as a "table of factors". He constructs the factorization as follows:

- The elements of the unit upper triangular matrix **U** are $u_{ij} = a_{ij}^{(i)}$, where $i < j$.
- The elements of the diagonal matrix **D** are $d_{ii} = \frac{1}{a_{ii}^{(i-1)}}$.
- The elements of the unit lower triangular matrix **L** are $l_{ij} = a_{ij}^{(j-1)}$, where $i < j$.

Figure 3 depicts the first two stages of Tinney's factorization scheme.

4.5 Numerical Instability During Factorization

Examining Equation 43 and Equation 44, you will observe that LU decomposition will fail when value the $a_{kk}^{(k)}$ (called the pivot element) is zero. In many applications, the possibility of a zero pivot is quite real and constitutes a serious impediment to the use of Gaussian elimination. This problem is compounded by the fact that Gaussian elimination is numerically unstable even if there are no zero pivot elements.

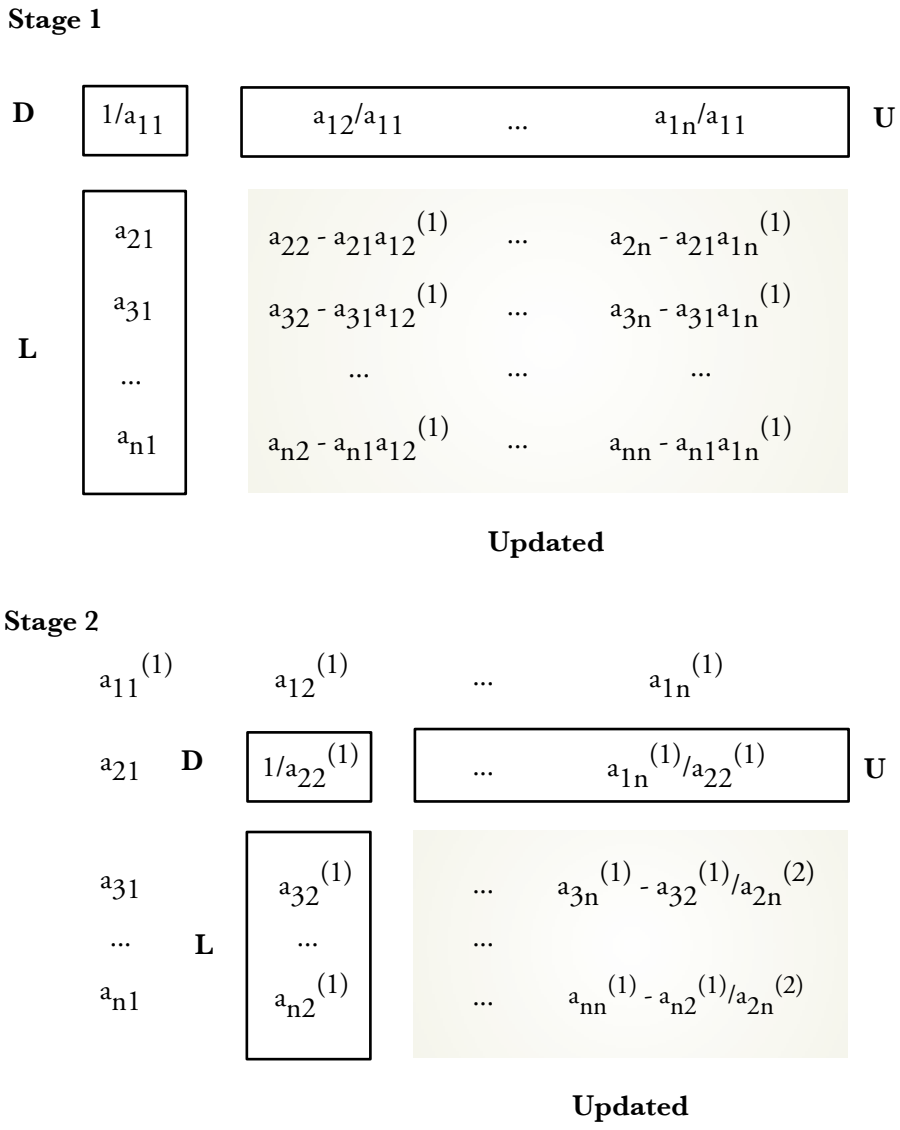
Numerical instability occurs when errors introduced by the finite precision representation of real numbers are of sufficient magnitude to swamp the true solution to a problem. In other words, a numerically unstable problem has a theoretical solution that may be unobtainable in finite precision arithmetic.

The other LU decomposition schemes examined in this section exhibit similar characteristics, e.g. instability is introduced by the division by u_{jj} in Equation 46 and l_{ii} in Equation 50.

4.6 Pivoting Strategies for Numerical Stability

A solution to the numerical instability of LU decomposition algorithms is obtained by interchanging the rows and columns of **A** to avoid zero (and other numerically unstable) pivot elements. These interchanges do not effect the solution to Equation 35 as long

Figure 3: Computational Sequence of Tinney's LDU Decomposition



Note: $a_{ij}^{(k)}$ is the k^{th} stage partial sum of a_{ij}.

as the permutations are logged and taken into account during the substitution process. The choice of pivot elements $a_{kk}^{(k)}$ is referred to as a pivot strategy. In the general case, there is no optimal pivot strategy. Two common heuristics are:

- At the k^{th} stage of the computation, choose the largest remaining element in $\mathbf{A}$ as the pivot. If pivoting has proceeded along the diagonal in stages 1 through $k-1$, this implies the next pivot should be the largest element $a_{ij}^{(k-1)}$ where $k \leq i \leq n$ and $k \leq j \leq n$. This strategy is referred to as complete pivoting.
- At the k^{th} stage of the computation, select the largest element in column k as the pivot. This strategy is referred to as partial pivoting.

Both procedures have good computational properties. Gaussian elimination with complete pivoting is numerically stable. In most practical applications, Gaussian elimination with partial pivoting has the same stability characteristics as complete pivoting. However, there are theoretical situations where partial pivoting strategies can become unstable.

Applications of current interest are diagonally dominant; therefore, algorithms which incorporate pivoting strategies for numerical stability are not examined in this document. For implementations of Gaussian elimination with complete and partial pivoting, see algorithms 4.4–1 and 4.4–2 of Golub and Van Loan (1983). For an implementation of Doolittle's method with scaled partial pivoting, see algorithm 3.5 of Conte and Boor (1972). Crout's method with scaled partial pivoting is implemented in section 2.3 of Press et al. (1988). Pivoting strategies which control element growth in sparse matrices are examined in Section 8.3 of this document.

The following sections present a brief introduction to the topic of pivoting to reduce numerical instability.

4.7 Diagonal Dominance and Pivoting

We will begin our discussion of pivoting by identifying a condition in which pivoting is unnecessary. The matrix $\mathbf{A}$ is row diagonally dominant when the following inequality holds.

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \text{where } i = 1, \cdots, n \tag{52}$$

The matrix $\mathbf{A}$ is column diagonally dominant when the following inequality holds.

$$|a_{jj}| > \sum_{j \neq i} |a_{ij}|, \text{where } j = 1, \cdots, n \tag{53}$$

If either of these conditions apply, the LU decomposition algorithms discussed in this document are numerically stable without pivoting.

4.8 Partial Pivoting

If a partial pivoting strategy is observed (pivoting is restricted to row interchanges), factorization produces matrices **L** and **U** which satisfy the following equation.

$$\mathbf{LU} = \mathbf{PA} \tag{54}$$

P is a permutation matrix that is derived as follows:

- **P** is initialized to **I**.
- Each row interchange that occurs during the decomposition of **A** causes a corresponding row swap in **P**.

Recalling the definition of a linear system of equations

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

and premultiplying both sides by **P**

$$\mathbf{PAx} = \mathbf{Pb}$$

Using Equation 54 to substitute for **PA** yields

$$\mathbf{LUx} = \mathbf{Pb} \tag{55}$$

Following the same train of logic used to derive equations Equation 40 and Equation 41 implies that a solution for **A** can be achieved by the sequential solution of two triangular systems.

$$\begin{aligned} \mathbf{y} &= \mathbf{Pb} \\ \mathbf{Lc} &= \mathbf{y} \\ \mathbf{Ux} &= \mathbf{c} \end{aligned} \tag{56}$$

Observe that the product **Pb** is computed before forward substitution begins. Computationally, this implies that **P** can be implemented as a mapping that is applied to **b** before substitution.

4.9 Complete Pivoting

If a complete pivoting strategy is observed (pivoting involves both row and column interchanges), factorization produces matrices **L** and **U** which satisfy the following equation.

$$\mathbf{LU} = \mathbf{PAQ} \tag{57}$$

where **P** is a row permutation matrix and **Q** is a column permutation matrix. **Q** is derived from column interchanges in the same way **P** is derived from row interchanges.

If **A** and its factors are related according to Equation 57, then Equation 35 can still be solved for **A** by the sequential solution of two triangular systems.

$$\mathbf{y} = \mathbf{P}\mathbf{b} \tag{58}$$
$$\mathbf{L}\mathbf{c} = \mathbf{y} \tag{59}$$
$$\mathbf{U}\mathbf{z} = \mathbf{c} \tag{60}$$
$$\mathbf{x} = \mathbf{Q}\mathbf{z} \tag{61}$$

Since Equation 56 and Equation 58 are identical, **P** can still be implemented as a mapping that is applied to **b** before substitution begins. Since Equation 61 computes the product **Qz** after back substitution is finished, **Q** can be implemented as a mapping that is applied to **A** following the substitution process.

If **A** is symmetric, pivoting for numerical stability may destroy the symmetry of the LU decomposition of **A**. For a symmetric factorization of **A**, matching row and column interchanges are required. In other words, pivoting must be complete and the permutation matrices must be related as follows:

$$\mathbf{Q} = \mathbf{P}^{\mathrm{T}}$$

4.10 Computational Complexity of Pivoting

Obviously, complete pivoting and partial pivoting differ substantially with regard to the computational effort required to determine the next pivot element. Complete pivoting on a dense, asymmetric matrix is an $O(n^3)$ operation requiring

$$\frac{2}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$$

floating point comparisons. Partial pivoting on the same matrix is an $O(n^2)$ operation requiring

$$\frac{n^2 + n}{2}$$

floating point comparisons.

4.11 Scaling Strategies

Some algorithms attempt to reduce the roundoff error generated during LU decomposition by preprocessing **A**. The most common roundoff control strategy is row scaling where each equation is normalized (so that its largest coefficient has a value of one) before a pivot is chosen. Pivoting strategies which employ scaling techniques usually are implemented in two stages:

Algorithm 4: Forward Substitution

for $i = 1, \cdots, n$
  $\alpha = b_i$
  for $j = 1, \cdots, i-1$
    $\alpha = \alpha - l_{ij} y_j$
  $y_i = \dfrac{\alpha}{l_{ii}}$

1. Scan the equations to determine a set of scale factors.
2. Choose the pivot elements taking the scale factors into account.

However, a word of caution is in order. Scaling techniques do not solve the fundamental roundoff problem of adding a large quantity to a small one. In fact, problems may arise where scaling techniques exacerbate rather than ameliorate roundoff error.

5 Solving Triangular Systems

A triangular system of equations is efficiently solved by a series of variable substitutions. By definition, there will always be at least one equation with a single unknown, one equation with only two unknowns (one of which will correspond to the unknown in the single variable equation), etc. This procedure can be formalized into two simple algorithms.

- A lower triangular system is solved by forward substitution.
- An upper triangular system is solved by backward substitution.

These operations are described with reference to the solution of systems of linear equations whose LU decomposition is known. Equation 41 (**Ly** = **b**) and Equation 40 (**Ux** = **y**) pose the problem in this context.

5.1 Forward Substitution

The equation **Ly** = **b** is solved by forward substitution as follows.

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} l_{ij} y_i}{l_{ii}}, \text{where } 1 \leq i \leq n \tag{62}$$

Algorithm 4 implements Equation 62.

If **L** is unit lower triangular, the division by l_{ii} is unnecessary (since l_{ii} is 1). Notice that the update to y_i is accumulated as an inner product in α.

Algorithm 5: Backward Substitution

for $i = n, \cdots, 1$
  $\alpha = y_i$
  for $j = i + 1, \cdots, n$
    $\alpha = \alpha - u_{ij} x_j$
  $x_i = \dfrac{\alpha}{u_{ii}}$

Algorithm 6: Forward Substitution - Outer Product

for $1 = 1, \cdots, n$
  $y_i = \dfrac{b_i}{l_{ii}}$
  for $k = i + 1, \cdots, n$
    $b_k = b_k - y_i l_{ki}$

5.2 Backward Substitution

The equation $\mathbf{y} = \mathbf{U}\mathbf{x}$ is solved by backward substitution as follows.

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} u_{ij} x_i}{u_{ii}}, \text{where } i = n, n-1, \cdots, 1 \tag{63}$$

Algorithm 5 implements Equation 63.

If $\mathbf{U}$ is unit upper triangular, the division by u_{ii} is unnecessary (since u_{ii} is 1). Notice that the update to x_i is accumulated as an inner product in α.

5.3 Outer Product Formulation

In Section 5.1 and Section 5.2 triangular systems are solved by techniques which use inner product accumulation on each row of $\mathbf{A}$. It is possible to formulate these algorithms in a manner that arrives at the solution through partial sum accumulations (also known as an outer product). Algorithm 6 solves lower triangular systems using an outer product formulation of forward substitution. You should observe that if b_i is zero, the i^{th} stage can be skipped. In this situation, y_i will also be zero and the term $y_i l_{ki}$ will not change any of the partial sums b_k.

Algorithm 7 solves upper triangular systems are using an outer product formulation of the backward substitution algorithm. Also observe that when y_i is zero, the i^{th} stage can

Algorithm 7: Back Substitution - Outer Product

for $i = 1, \cdots, n$
  $x_i = \dfrac{y_i}{u_{ii}}$
  for $k = i - 1, \cdots, 1$
    $y_k = y_k - x_i u_{ki}$

be skipped. In this situation, x_i will also be zero and the term $x_i u_{ki}$ will not change any of the partial sums y_k.

George and Liu (1981) examine outer product solutions of triangular systems as do Tinney, Brandwajn, and Chan (1985).

6 Factor Update

If the LU decomposition of the matrix **A** exists and the factorization of a related matrix

$$\mathbf{A}' = \mathbf{A} + \mathbf{\Delta A} \tag{64}$$

is needed, it is sometimes advantageous to compute the factorization of $\mathbf{A}'$ by modifying the factors of **A** rather than explicitly decomposing $\mathbf{A}'$. Implementations of this factor update operation should have the following properties:

- Arithmetic is minimized,
- Numerical stability is maintained, and
- Sparsity is preserved.

The current discussion outlines procedures for updating the factors of **A** following a rank one modification. A rank one modification of **A** is defined as

$$\mathbf{A}' = \mathbf{A} + \alpha \mathbf{y} \mathbf{z}^{\mathbf{T}} \tag{65}$$

where α is a scalar and the vectors **y** and $\mathbf{z}^{\mathbf{T}}$ are dimensionally correct. The terminology comes from the observation that the product $\alpha \mathbf{z}^{\mathbf{T}}$ is a matrix whose rank is one.

Computationally, a rank one factor update to a dense matrix is an $O(n^2)$ operation. Recall that decomposing a matrix from scratch is $O(n^3)$.

6.1 LDU Factor Update

Algorithm 8 follows the lead of our sources (see Section 6.3 for details) and implements a technique for updating the factors **L**, **D**, and **U** of **A** following a rank one change to **A**.

Algorithm 8: LDU Factor Update

for $i = 1, \cdots, n$
  $\delta = d_i$
  $p = y_i$
  $q = z_i$
  $d_i = d_i + \alpha pq$
  $\beta_1 = \dfrac{\alpha p}{d_i}$
  $\beta_2 = \dfrac{\alpha q}{d_i}$
  $\alpha = \dfrac{\alpha \delta}{d_i}$
  for $j = i + 1, \cdots, n$
    $y_j = y_j - p l_{ji}$
    $z_j = z_j - q u_{ij}$
    $l_{ji} = l_{ji} + \beta_1 y_j$
    $u_{ij} = u_{ij} + \beta_2 y_j$

The scalar α and the vectors $\mathbf{y}$ and $\mathbf{z}^{\mathbf{T}}$ are destroyed by this procedure. The factors of $\mathbf{A}$ are overwritten by their new values.

The outer loop of Algorithm 8 does not have to begin at one unless $\mathbf{y}$ is full. If $\mathbf{y}$ has any leading zeros, the initial value of i should be the index of y_i the first nonzero element of $\mathbf{y}$. If there is no *a priori* information about the structure of $\mathbf{y}$ but there is a high probability of leading zeros, testing y_i for zero at the beginning of the loop might save a lot of work. However, you must remember to cancel the test as soon as a nonzero y_i is encountered.

6.2 LU Factor Update

Similar algorithms exist for updating LU decompositions of $\mathbf{A}$. If $\mathbf{U}$ is upper triangular and $\mathbf{L}$ is unit lower triangular, an element u_{ij} from $\mathbf{U}$ is related to the elements u'_{ij} and d'_i of the LDU decomposition as follows.

$$u_{ij} = u'_{ij} d'_i \tag{66}$$

The recurrence relations of the inner loop of Algorithm 8 must change to reflect this relationship. The following statement updates the $\mathbf{z}$ vector for a unit upper triangular $\mathbf{U}$.

$$z_j = z_j - q u'_{ij} \tag{67}$$

Algorithm 9: LU Factor Update

for $i = 1, \cdots, n$
$\delta = u_{ii}$
$p = y_i$
$q = z_i$
$u_{ii} = u_{ii} + \alpha pq$
$\beta_1 = \dfrac{\alpha}{u_{ii}}$
$\beta_2 = \alpha q$
$q = \dfrac{q}{\delta}$
$\delta = \dfrac{u_{ii}}{\delta}$
$\alpha = \dfrac{\alpha}{\delta}$
for $j = i + 1, \cdots, n$
$y_j = y_j - pl_{ji}$
$z_j = z_j - qu_{ij}$
$l_{ji} = l_{ji} + \beta_1 y_j$
$u_{ij} = \delta u_{ij} + \beta_2 z_j$

If **U** is upper triangular, the statement becomes

$$z_j = z_j - p\frac{u_{ij}}{\delta} \tag{68}$$

where δ is the value of u_{ii} before it was changed during stage i of the procedure. Along the same lines, the factor update statement

$$u'_{ij} = u'_{ij} + \beta_2 z_j \tag{69}$$

becomes

$$\frac{u_{ij}}{u_{ii}} = \frac{u_{ij}}{\delta} + \beta_2 z_j \tag{70}$$

Solving for the updated value of u_{ij} yields

$$u_{ij} = u_{ii}(\frac{u_{ij}}{\delta} + \beta_2 z_j) \tag{71}$$

Taking these observations into consideration and pulling operations on constants out of the inner loop, Algorithm 9 updates **U** based on a rank one change to **A**.

If **U** is unit upper triangular and **L** is lower triangular, a similar algorithm is derived from the observation that l_{ij} of **L** and l'_{ij}, d'_i of the LDU decomposition are related as follows.

$$l_{ij} = l'_{ij} d'_j \tag{72}$$

The resulting algorithm deviates from Algorithm 8 in a manner that parallels Algorithm 9.

6.3 Additional Considerations

The algorithms presented in Section 6.1 and Section 6.2 are based on the work of Bennett (1965). The nomenclature is similar to that of Gill et al. (1974). These citations describe procedures for updating the factors of an LDU decomposition.

The procedure described by Bennett (1965) is more general than the algorithms described in this section in that it applies to rank m changes to $\mathbf{A}$. However, decomposing a rank m change into m rank one changes and applying the current algorithms has the same complexity as Bennett's process and saves a little array space. Gill et al. (1974) state that Bennett's algorithm is theoretically unstable unless $\mathbf{L} = \mathbf{U}^{\mathrm{T}}$ and $\mathbf{y} = \mathbf{z}$. In practice, Bennett's algorithm has proven to be stable for many physical problems with reasonable values of α, $\mathbf{y}$, and $\mathbf{z}$. The algorithm rarely exhibits instability when it is applied to diagonally dominant matrices where pivoting is not required. Gill et al. (1974) describe alternate algorithms for situations where stability problems arise.

Hager (1989) provides a good overview of approaches to the problem of updating the inverse of a matrix and describes practical areas in which the problem arises. Chan and Brandwajn (1986) examine applications in network analysis.

7 Symmetric Matrices

Recall that an $n \times n$ symmetric matrix $\mathbf{A}$ is its own transpose

$$\mathbf{A} = \mathbf{A}^{\mathrm{T}}$$

This being the case, the elements of $\mathbf{A}$ are described by the following relationship

$$a_{ij} = a_{ji}, \text{ for all } i, j \tag{73}$$

7.1 LDU Decomposition of Symmetric Matrices

If $\mathbf{A}$ is symmetric, its LDU decomposition is symmetric, i.e.

$$\mathbf{L} = \mathbf{U}^{\mathrm{T}} \tag{74}$$

and

$$\mathbf{U} = \mathbf{L}^{\mathrm{T}} \tag{75}$$

For this reason, the LDU decomposition of a symmetric matrix is sometimes referred to as an LDL^T decomposition. The elements $\mathbf{L}$ and $\mathbf{U}$ of the LDU decomposition of a symmetric matrix are related as follows.

$$l_{ij} = u_{ji}, \text{where } i \neq j \tag{76}$$

7.2 LU Decomposition of Symmetric Matrices

Given the symmetric structure of the LDU factors of a symmetric matrix (see Section 7.1) and the common use of LU factorization in the analysis of linear systems, it is constructive to develop expressions that relate an explicit LU decomposition to an implicit LDU factorization. In subsequent sections of this document, they will prove useful in deriving symmetric variants of the algorithms discussed in Section 4 and Section 5.

In other words, the symmetric factorization algorithms discussed in this document assume an LU decomposition exists (or is to be computed) such that

- Its existing (explicit) factorization (either $\mathbf{L}$ or $\mathbf{U}$) is triangular, and
- The desired (implicit) factorization is unit triangular.

This implies that algorithms which deal with an explicit set of lower triangular factors, call them $\overline{\mathbf{L}}$, will associate the factors of an implicit LDU decomposition as follows

$$\overline{\mathbf{L}} = \mathbf{LD} \tag{77}$$

or

$$\mathbf{L} = \overline{\mathbf{L}}\mathbf{D}^{-1}$$

Substituting for $\mathbf{L}$ based on Equation 74 yields

$$\mathbf{U}^{\mathbf{T}} = \overline{\mathbf{L}}\mathbf{D}^{-1} \tag{78}$$

Recalling that the inverse of a diagonal matrix is the arithmetic inverse of each element and taking the product yields

$$u_{ji} = \frac{\bar{l}_{ij}}{d_{ii}}$$

Since $d_{ii} = \bar{l}_{ii}$,

$$u_{ij} = \frac{\bar{l}_{ji}}{\bar{l}_{jj}} \tag{79}$$

In a similar vein, algorithms that deal with an explicit set of upper triangular factors, call them $\overline{\mathbf{U}}$, will associate the factors of an LDU decomposition as follows.

$$\overline{\mathbf{U}} = \mathbf{DU} \tag{80}$$

This association yields the following relationship between the explicit factors $\overline{\mathbf{U}}$ and implicit factors $\mathbf{L}$.

$$l_{ij} = \frac{\overline{u}_{ji}}{\overline{u}_{jj}} \tag{81}$$

These observations show that it is only necessary to compute and store $\overline{\mathbf{L}}$ or $\overline{\mathbf{U}}$ during the LU factorization of a symmetric matrix. This halves the arithmetic required during the factorization procedure. However, symmetry does not reduce the work required during forward and backward substitution.

7.3 Symmetric Matrix Data Structures

Recognizing the special character of symmetric matrices can save time and storage during the solution of linear systems. More specifically, a dense matrix requires storage for n^2 elements. A symmetric matrix can be stored in about half the space, $\frac{n^2+n}{2}$ elements. Only the upper (or lower) triangular portion of $\mathbf{A}$ has to be explicitly stored. The implicit portions of $\mathbf{A}$ can be retrieved using Equation 73. An efficient data structure for storing dense, symmetric matrices is a simple linear array. If the upper triangular portion of $\mathbf{A}$ is retained, the array is organized in the following manner.

$$\mathbf{A} = (a_{11}, a_{12}, \cdots, a_{1n}, a_{21}, \cdots, a_{2n}, \cdots, a_{nn}) \tag{82}$$

The element a_{ij} is retrieved from the linear array by the following indexing rule.

$$a_{ij} = \mathbf{a}[(i-1)(n) - (i-1)i/2 + j] \tag{83}$$

If array and matrix indexing is zero based (as in the C programming language), the subscripting rule becomes

$$a_{ij} = \mathbf{a}[in - (i-1)i/2 + j] \tag{84}$$

If the lower triangular portion of $\mathbf{A}$ is retained, the linear array is organized as follows.

$$\mathbf{A} = (a_{11}, a_{21}, a_{22}, a_{31}, \cdots, a_{n1}, a_{n2}, \cdots, a_{nn}) \tag{85}$$

The element a_{ij} is retrieved from the linear array by the following indexing rule.

$$a_{ij} = \mathbf{a}[i(i-1)/2 + j] \tag{86}$$

If array and matrix subscripts are zero based, Equation 86 becomes

$$a_{ij} = \mathbf{a}[i(i+1)/2 + j] \tag{87}$$

You will observe that the dimension of $\mathbf{A}$ does not enter the indexing calculation when its lower triangular portion is retained. The indexing equations are implemented most efficiently by replacing division by two with a right shift.

Algorithm 10: Doolittle's Method - Symmetric Implementation

for $i = 1, \cdots, n$
  for $j = 1, \cdots, i-1$
    $w_j = \dfrac{a_{ji}}{a_{jj}}$
  for $j = i, \cdots, n$
    $\alpha = a_{ij}$
    for $p = 1, \cdots, i-1$
      $\alpha = \alpha - a_{ip} w_p$
    $a_{ij} = \alpha$

Algorithm 11: Doolittle's Method - Symmetric, Array Based

for $i = 0, \cdots, n-1$
  for $j = 0, \cdots, i-1$
    $\mathbf{w}[j] = \dfrac{\mathbf{a}[jn-j(j+1)/2+i]}{\mathbf{a}[jn-j(j+1)/2+j]}$
  for $j = i, \cdots, n-1$
    $\alpha = \mathbf{a}[in-i(i+1)/2+j]$
    for $p = 1, \cdots, i-1$
      $\alpha = \alpha - \mathbf{a}[in-i(i+1)/2+p] \cdot \mathbf{w}[p]$
    $\mathbf{a}[in-i(i+1)/2+j] = \alpha$

7.4 Doolittle's Method for Symmetric Matrices

If $\mathbf{A}$ is a symmetric $n \times n$ matrix, Algorithm 10 computes – one row at a time – the upper triangular matrix $\mathbf{U}$ that results from a Doolittle decomposition. The upper triangular portion of $\mathbf{A}$ is overwritten by $\mathbf{U}$.

The algorithm uses a working vector $\mathbf{w}$ of length n to construct the relevant portion of row i from $\mathbf{L}$ at each stage of the factorization. Elements from $\mathbf{L}$ are derived using Equation 81.

If the upper triangular portion of $\mathbf{A}$ is stored in a linear array, Algorithm 11 results in the same factorization (assuming zero based subscripting). The upper triangular portion of $\mathbf{A}$ is overwritten by $\mathbf{U}$.

For the general implementation of Doolittle's method, see Section 4.2.

Algorithm 12: Crout's Method - Symmetric Implementation

for $j = 1, \cdots, n$
  for $i = 1, \cdots, j-1$
    $w_i = \dfrac{a_{ji}}{a_{ii}}$
  for $i = j, \cdots, n$
    $\alpha = a_{ij}$
    for $p = 1, \cdots, j-1$
      $\alpha = \alpha - a_{ip} w_p$
    $a_{ij} = \alpha$

Algorithm 13: Crout's Method - Symmetric, Array Based

for $j = 0, \cdots, n-1$
  for $i = 0, \cdots, j-1$
    $\mathbf{w[j]} = \dfrac{\mathbf{a[jn-j(j+1)/2+i]}}{\mathbf{a[jn-j(j+1)/2+j]}}$
  for $i = j, \cdots, n-1$
    $\alpha = \mathbf{a[i(i+1)/2+j]}$
    for $p = 1, \cdots, i-1$
      $\alpha = \alpha$ - $\mathbf{a[i(i+1)/2+p]} \cdot \mathbf{w[p]}$
    $\mathbf{a[i(i+1)/2+j]} = \alpha$

7.5 Crout's Method for Symmetric Matrices

If $\mathbf{A}$ is a symmetric $n \times n$ matrix, Algorithm 12 computes – one column at a time – the lower triangular matrix $\mathbf{L}$ that results from a Crout decomposition. The lower triangular portion of $\mathbf{A}$ is overwritten by $\mathbf{L}$.

The algorithm uses a working vector $\mathbf{w}$ of length n to construct the relevant portion of column j from $\mathbf{U}$ at each stage of the factorization. Elements from $\mathbf{U}$ are derived using Equation 79.

If the lower triangular portion of $\mathbf{A}$ is stored in a linear array, Algorithm 13 results in the same factorization (assuming zero based subscripting). The algorithm overwrites $\mathbf{A}$ with $\mathbf{L}$.

For the general implementation of Crout's method, see Section 4.3.

Algorithm 14: Symmetric Forward Substitution via Upper Triangular Factors

for $i = 1, \cdots, n$
  $\alpha = b_i$
  for $j = 1, \cdots, i-1$
    $\alpha = \alpha - \frac{u_{ji}}{u_{jj}} y_j$
  $y_i = \alpha$

7.6 Forward Substitution for Symmetric Systems

Symmetry reduces the amount of work required to decompose symmetric systems into triangular factors. It does not reduce the work required to actually solve the system from an existing triangular factorization. Implementing forward substitution for a symmetric decomposition boils down to making sure implicit data (i.e. the portion of the symmetric factorization that is not not physically stored) is correctly derived from the explicitly stored data. See Section 7.2 for a discussion of implicit data in symmetric LU decompositions.

7.6.1 Forward Substitution Using Lower Triangular Factors

Forward substitution solves lower triangular systems. When $\mathbf{L}$ is available, the symmetric and asymmetric solution algorithms are identical. See Section 5.1 for the general implementation of forward substitution. If $\mathbf{L}$ is stored in a linear array, use Equation 86 or Equation 87 for indexing.

7.6.2 Forward Substitution Using Upper Triangular Factors

The case in which $\mathbf{U}$ is the explicit factorization is examined more closely. If $\mathbf{U}$ is an $n \times n$ matrix containing the upper triangular factors of a symmetric matrix, then $\mathbf{L}$ is unit lower triangular and obtained from $\mathbf{U}$ via Equation 81 with l_{jj} being one. Substituting Equation 81 into Equation 62 yields:

$$y_i = b_i - \sum_{j=i+1}^{n} \frac{u_{ji} y_j}{u_{jj}}, \text{where } 1 \leq i \leq n \tag{88}$$

Algorithm 14 implements Equation 88, i.e. the inner product formulation of forward substitution for symmetric systems whose upper triangular factors are available.

If $\mathbf{U}$ is stored in a linear array with zero based indexing, the inner product formulation of forward substitution is implemented by Algorithm 15.

Algorithm 15: Symmetric Forward Substitution using **U** with Array Storage

for $i = 0, \cdots, n-1$
  $\alpha = \mathbf{b}[\mathrm{i}]$
  for $j = 0, \cdots, i-1$
    $\alpha = \alpha - \dfrac{\mathbf{u}[\mathrm{jn-j(j+1)/2+i}]}{\mathbf{u}[\mathrm{jn-j(j+1)/2+j}]} \cdot \mathbf{y}[\mathrm{j}]$
  $\mathbf{y}[\mathrm{i}] = \alpha$

Algorithm 16: Symmetric Forward Substitution using **U**, Outer Product

for $i = 1, \cdots, n$
  $y_i = b_i$
  $\alpha = \dfrac{y_i}{u_{ii}}$
  for $k = i+1, \cdots, n$
    $b_k = b_k - \alpha u_{ik}$

Algorithm 16 implements the outer product formulation of forward substitution for symmetric systems whose upper triangular factors are available.

This differs from the asymmetric implementation in that

$$l_{ii} = 1 \text{ and } l_{ki} = \frac{u_{ik}}{u_{ii}}, \text{where } i \neq k \tag{89}$$

Therefore, the initial division by l_{ii} is omitted and the division by u_{ii} is pulled out of the k loop. The outer product formulation of forward substitution where **U** is stored in a linear array with zero based indexing is realized by Algorithm 17.

7.7 Backward Substitution for Symmetric Systems

As stated previously, symmetry reduces the amount of work required to decompose symmetric systems into triangular factors. It does not reduce the work required to

Algorithm 17: Symmetric Forward Substitution using **U**, Outer Product, Array

for $i = 0, \cdots, n-1$
  $\mathbf{y}[\mathrm{i}] = \mathbf{b}[\mathrm{i}]$
  $\alpha = \dfrac{\mathbf{y}[\mathrm{i}]}{\mathbf{u}[\mathrm{in-i(i+1)/2+i}]}$
  for $k = i+1, \cdots, n-1$
    $\mathbf{b}[\mathrm{k}] = \mathbf{b}[\mathrm{k}] - \alpha \cdot \mathbf{u}[\mathrm{in-i(i+1)/2+k}]$

Algorithm 18: Symmetric Back Substitution using Lower Triangular Factors

for $i = 1, \cdots, n$
  $\alpha = y_i$
  for $j = 1, \cdots, i-1$
    $\alpha = \alpha - \frac{l_{ji}}{l_{jj}} x_j$
  $x_i = \alpha$

actually solve the system from an existing triangular factorization. Implementing backward substitution for a symmetric decomposition reduces to making sure that implicit data (i.e. the portion of the symmetric factorization that is not not physically stored) is correctly derived from the explicitly stored data. See Section 7.2 for a discussion of implicit data in symmetric LU decompositions.

7.7.1 Back Substitution Using Upper Triangular Factors

Backward substitution solves upper triangular systems. When $\mathbf{U}$ is available, the symmetric and asymmetric solution algorithms are identical. See Section 5.2 for the general implementation of backward substitution. If $\mathbf{U}$ is stored in a linear array, use Equation 86 or Equation 87 for indexing.

7.7.2 Back Substitution Using Lower Triangular Factors

The case in which $\mathbf{L}$ is the explicit factorization merits further attention. If $\mathbf{L}$ is an $n \times n$ matrix containing the upper triangular factors of a symmetric matrix, then $\mathbf{U}$ is unit lower triangular and obtained from $\mathbf{L}$ via Equation 79 with u_{ii} being one. Substituting Equation 79 into Equation 63 yields:

$$x_i = y_i - \sum_{j=i+1}^{n} \frac{l_{ji} x_j}{l_{jj}}, \text{where } i = n, n-1, \cdots, 1 \tag{90}$$

Algorithm 18 implements this inner product formulation of backward substitution for symmetric systems whose lower triangular factors are available.

If $\mathbf{L}$ is stored in a linear array with zero based indexing, the inner product formulation of back substitution is stated in Algorithm 19.

Algorithm 19: Symmetric Backward Substitution using **L** with Array Storage

for $i = n-1, \cdots, 0$
  $\alpha = \mathbf{y}[i]$
  for $j = i+1, \cdots, n-1$
    $\alpha = \alpha - \dfrac{\mathbf{l}[jn\text{-}j(j+1)/2+i]}{\mathbf{l}[jn\text{-}j(j+1)/2+j])} \cdot \mathbf{x}[j]$
  $\mathbf{x}[i] = \alpha$

7.8 Symmetric Factor Update

If the LU decomposition of the $n \times n$ symmetric matrix $\mathbf{A}$ exists and the factorization of a related matrix

$$\mathbf{A}' = \mathbf{A} + \mathbf{\Delta A}$$

is desired, factor update is often the procedure of choice.

Section 6 examines factor update techniques for dense, asymmetric matrices. The current section examines techniques that exploit computational efficiencies introduced by symmetry. Symmetry reduces the work required to update the factorization of $\mathbf{A}$ by half, just as it reduces the work required to decompose $\mathbf{A}$ in the first place.

More specifically, the current section examines procedures for updating the factors of $\mathbf{A}$ following a symmetric rank one modification

$$\mathbf{A}' = \mathbf{A} + \alpha \mathbf{y}\mathbf{y}^{\mathrm{T}}$$

where α is a scalar and $\mathbf{y}$ is an n vector.

7.8.1 Symmetric LDU Factor Update

Algorithm C1 of Gill et al. (1974) updates the factors $\mathbf{L}$ and $\mathbf{D}$ of a LDU decomposition of $\mathbf{A}$. The algorithm assumes that the upper triangular factors $\mathbf{U}$ are implicit. Algorithm 20 mimics Algorithm C1. The scalar α and the $\mathbf{y}$ vector are destroyed by this procedure. The factors of $\mathbf{A}$ are overwritten by their new values.

See Section 6.1 for a discussion of updating asymmetric LDU factorizations following rank one changes to a matrix.

7.8.2 Symmetric LU Factor Update

If the LU decomposition of the symmetric matrix $\mathbf{A}$ exists and $\mathbf{U}$ is stored explicitly, the recurrence relations of the inner loop of Algorithm 20 must change. Following a

Algorithm 20: Symmetric LDU Factor Update

for $j = 1, \cdots, n$
  $\delta = d_j$
  $p = y_j$
  $d_j = d_j + \alpha p^2$
  $\beta = \dfrac{\alpha p}{d_j}$
  $\alpha = \dfrac{\alpha \delta}{d_j}$
  for $i = j + 1, \cdots, n$
    $y_i = y_i - p l_{ij}$
    $l_{ij} = l_{ij} + \beta y_i$

Algorithm 21: Symmetric LU Factor Update

for $i = 1, \cdots, n$
  $\delta = u_{ii}$
  $p = y_i$
  $u_{ii} = u_{ii} + \alpha p^2$
  $\beta = \alpha p$
  $p = \dfrac{p}{\delta}$
  $\delta = \dfrac{u_{ii}}{\delta}$
  $\alpha = \dfrac{\alpha}{\delta}$
  for $j = i + 1, \cdots, n$
    $y_j = y_j - p u_{ij}$
    $u_{ij} = \delta u_{ij} + \beta y_y$

train of thought similar to the derivation of Algorithm 9 (see Section 6.2 for details) results in Algorithm 21 which updates **U** based on a rank one change to **A**.

If **U** is maintained in a zero-based linear array, Algorithm 21 changes in the normal manner, that is

1. The double subscript notation is replaced by the indexing rule defined in Equation 87.
2. The outer loop counter *i* ranges from zero to *n-1*.
3. The inner loop counter *j* ranges from *i+1* to *n-1*.

The outer loops of the symmetric factor update algorithms do not have to begin at one unless **y** is full. If **U** has any leading zeros, the initial value of *i* (or *j* in Algorithm 20)

should be the index of y_i the first nonzero element of $\mathbf{y}$. If there is no *a priori* information about the structure of $\mathbf{y}$ but there is a high probability of leading zeros, testing y_i for zero at the beginning of the loop might save a lot of work. However, you must remember to suspend the test as soon as the first nonzero value of y_i is encountered.

For a fuller discussion of the derivation and implementation of LU factor update, see Section 6.2.

8 Sparse Matrices

The preceding sections examined dense matrix algorithms for solving systems of linear equations. It was seen that significant savings in storage and computation is achieved by exploiting the structure of symmetric matrices. An even more dramatic performance gain is possible by exploiting the sparsity intrinsic to many classes of large systems. Sparse matrix algorithms are based on the simple concept of avoiding the unnecessary storage of zeros and unnecessary arithmetic associated with zeros (such as multiplication by zero or addition of zero). Recognizing and taking advantage of sparsity often permits the solution of problems that are otherwise computationally intractable. Practical examples provided by Tinney and Hart (1972) suggest that in the analysis of large power system networks the use of sparse matrix algorithms makes both the storage and computational requirements approximately linear with respect to the size of the network. In other words, data storage is reduced from an $O(n^2)$problem to an $O(n)$ problem and computational complexity diminishes from$O(n^3)$ to $O(n)$.

8.1 Sparse Matrix Methodology

Any matrix with a significant number of zero-valued elements is referred to as a sparse matrix. The meaning of "significant" in the preceding definition is rather vague. It is pinned down (in a circular way) by defining a sparse matrix to be a matrix with enough zeros to benefit from the use of sparsity techniques. The intent of this definition is to emphasize that there is a computational overhead required by sparse matrix procedures. If the degree of sparsity in a matrix compensates for the algorithmic overhead, sparsity techniques should be employed. Otherwise, dense matrix algorithms should be utilized. This argument simply restates a fundamental rule of numerical analysis, *a priori* information concerning the nature of a problem tends to result in more efficient solution techniques.

The next few sections will explore the application of sparsity techniques to the solution of large systems of linear equations. The standard approach is to break the solution into three phases:

1. *Analyze*. Determine an ordering of the equations such that the LU decomposition will retain as much sparsity as possible. This problem has been shown to be N-P

complete (i.e. the optimal solution can not be efficiently determined). However, a number of satisfactory heuristics are available. The analysis phase of the solution usually produces a complete definition of the sparsity pattern that will result when the LU decomposition is computed.

2. *Factor*. Compute the LU decomposition.
3. *Solve*. Use the LU decomposition to compute a solution to the system of equations (i.e. perform forward and backward substitution).

The degree to which these phases are distinct depends on the implementation.

8.2 Abstract Data Types for Sparse Matrices

The traditional implementation of sparse matrix techniques in engineering and scientific analysis is heavily reminiscent of its roots in the static data structures of FORTRAN. Descriptions of these data structures are provided by Duff, Erisman, and Reid (1986), Tinney and Hart (1972) among others. The current implementation takes a different tack. Sparse matrix algorithms are described using an abstract data type paradigm. That is, data sets and operators are specified, but the actual data structures used to implement them are left undefined. Any data structure that efficiently satisfies the constraints imposed in this section is suited for the job.

All signals emitted by the operators defined in this section are used to navigate through data, not to indicate errors. Error processing is intentionally omitted from the algorithms appearing in this document. The intent is to avoid clutter that obscures the nature of the algorithms.

8.2.1 Sparse Matrix

A sparse matrix, **A**, is stored in a dynamic data structure that locates an element a_{ij} based on its row index i and column index j. The following operations are supported on **A**:

- `Insert` adds an arbitrary element a_{ij} to **A**. If a_{ij} does not already exist, `insert` signals a successful insertion. In subsequent algorithms, insertion of element $a_{i,j}$ into a sparse matrix is represented as

  `insert(` **A**, i, j, a `)`

- `Get` retrieves an arbitrary element a_{ij} from **A**. When a_{ij} is an element of **A**, `get` signals a successful lookup. In subsequent algorithms, retrieval of element $a_{i,j}$ from a sparse matrix is represented as

  `get(` **A**, i, j, a `)`

- `Scan` permits sequential access to the nonzero entries of row i of **A**. Row scans are bounded. More specifically, a row scan finds all nonzero entries a_{ij} in row i

of **A** such that $j_{min} \leq j \leq j_{max}$.When `scan` finds a_{ij} its column index j is returned. When the scan has exhausted all entries in its range, a *finished* signal is emitted. In subsequent algorithms, iterating row i of a sparse matrix is represented as

`scan`(**A**, i, j_{min}, j_{max}, a)

A `scan` has two support operations, `push_scan` and `pop_scan`, which permit the nesting of scans.

`Push_scan` suspends the scan at its current position.

`Pop_scan` resumes a suspended scan.

- `Put` updates the value an arbitrary element a_{ij} of **A**. In subsequent algorithms, updating element $a_{i,j}$ of a sparse matrix is represented as

  `put`(**A**, i, j, a)

In the context of the preceding function prototypes, parameters **A** and a can be thought of as C/C++ pointers or references. Other arguments can be viewed as integers.

The sparse matrix algorithms assume that operations that read the data structure (`get` and `scan`) make the designated element a_{ij} of **A** available in a buffer. Operations that update a_{ij} (`insert` and `put`) do so based on the current contents of the communication buffer.The buffer can be conceptualized as a C/C++ struct. For sparse matrices, the structure always contains at least one field: an element of the matrix. To avoid clutter in the algorithms, this "element" field is implicit. The assignment

$$\delta = a$$

acutally represents

$$\delta = a.element$$

Should other fields of the structure become relevant, they are represented explicitly.

Section 9.1 examines one possible realization of the sparse matrix data type.

8.2.2 Adjacency List

An adjacency list, A, is a data type for representing adjacency relationships of the sparse graph $G = (V, E)$. An adjacency list is typically stored in a dynamic data structure that identifies the edge from vertex i to vertex j as an ordered pair of vertex labels (i,j). Descriptive information is usually associated with each edge. Basic concepts of adjacency lists are reviewed in Section 2.3 (Representing Sparse Graphs as Adjaceny Lists) of the companion monograph *Graph Algorithms*.

Since both adjacency lists and sparse matrices represent sparse networks, it should come as no surprise that they require a similar set of operations. More specifically, the following operations are supported on an adjacency list A:

- `Insert` adds an arbitrary edge $e_{i,j}$ to A. If edge $e_{i,j}$ is not already in the list, `insert` signals a successful insertion. In subsequent algorithms, insertion of edge $e_{i,j}$ into an adjacency list is represented as

  `insert`(A, i, j, e)

- `Get` retrieves an arbitrary edge $e_{i,j}$ from A. When edge $e_{i,j}$ is in A, `get` signals a successful lookup. In subsequent algorithms, retrieval of of edge $e_{i,j}$ from an adjacency list is represented as

  `get`(A, i, j, e)

- `Iterate` permits sequential access to all edges incident upon vertex i. Vertex scans are bounded. More specifically, a vertex scan finds all edges $e_{i,j}$ such that $j_{min} \leq j \leq j_{max}$. When `iterate` finds edge $e_{i,j}$, it returns j. When a vertex scan has exhausted all entries in its range, a *finished* signal is emitted. In subsequent algorithms, iterating the edges associated with vertex i is represented as

  `iterate`($A, i, j_{min}, j_{max}, e$)

  A `iterate` has two support operations, `push_iteratation` and `pop_iteration`, which permit nesting of the operation.

  `Push_iteratation` saves the current state of an iteration process.

  `Pop_iteratation` restores the most recently suspended iteration state.

- `Put` updates the information associated with an arbitrary edge $e_{i,j}$ in A. In subsequent algorithms, updating edge $e_{i,j}$ of an adjacency list is represented as

  `put`(A, i, j, e)

In the context of the preceding function prototypes, parameters A and e can be thought of as C/C++ pointers or references. Other arguments can be viewed as integers.

The algorithms assume that read operations (`get` and `scan`) make edge information available in a buffer (this buffer is usually denoted by the symbol e). Update operations (`insert` and `put`) modify the information associated with an edge based on the current contents of the communication buffer. The buffer can be conceptualized as a C/C++ struct. Any of the adjacency list buffer fields required by an algorithm are explicitly referenced, ie. *e.fillup* represents the "fillup" edge buffer field.

Algorithms for implementing adjacency lists are examined in Section 4 (Creating Adjacency Lists) of the companion monograph *Graph Algorithms*.

8.2.3 Reduced Graph

A reduced graph, $G' = (V', E')$, is a data structure that supports the systematic elimination of all vertices from the graph $G = (V, E)$. The vertices of the reduced graph are denoted as $V'(G')$ and its edges as $E'(G')$. A crucial attribute of the reduced graph is efficient identification of the vertex in $V'(G')$ with the minimum degree.

A reduced graph supports the following operations:

- `Increase_degree` increases the degree of vertex v in $V'(G')$ by one.
- `Decrease_degree` decreases the degree of vertex v in $V'(G')$ by one.
- `In_graph` tests to see whether vertex v is in $V'(G')$.
- `Minimum_degree` finds the vertex v in $V'(G')$ with the smallest degree.
- `Remove` excises vertex v from $V'(G')$.

Implementation of reduced graph modeling is examined in detail by Section 8.2 (Eliminating Many Vertices), Section 8.3 (Initializing Minimum Degree Vertex Tracking), and Section 8.4 (Maintaining the Reduced Graph) of *Graph Algorithms*.

8.2.4 List

A simple list L is an ordered set of elements. If the set $\{l_1, ..., l_i, l_{i+1}, ..., l_n\}$ represents L, then the list contains n elements. Element l_1 is the first item on the list and l_n is the last item on the list. Element l_i precedes l_{i+1} and element l_{i+1} follows l_i. Element l_i is at position i in L. Descriptive information may accompany each item on a list. Lists associated with matrix algorithms support the following operations:

- `Link` adds an element x to a list at position i. Inserting element x into the list at position i results an an updated list: $\{l_1, ..., l_i, l_{i+1}, ..., l_n\}$ An insertion at position *eol* appends x to the end of the list.
- `Unlink` removes the element at position i from the list. Deleting element i results in the list $\{l_1, ..., l_i, l_{i+1}, ..., l_n\}$.
- `Find` looks for an element on the list and returns its position. If the element is not a member of the list, *eol* is returned.
- `First` returns the position of the first item on the list. When the list is empty, *eol* is returned.
- `Next` returns position i+1 on the list if position i is provided. If l_i is the last item on the list, *eol* is returned.
- `Prev` returns position i-1 on the list if position i is provided. If i is one, *eol* is returned.

A linked list refers to a list implementation that does not require its members to reside in contiguous storage locations. In this environment, an efficient implementation of the `prev` operator dictates the use of a doubly linked list.

Communicating with a simple list is analogous to adjacency list communication. Read operations (`find`, `first`, `next`, and `prev`) make list information available in a buffer. Update operations (`link`, `unlink`) modify the list based on the current contents of the buffer.

8.2.5 Mapping

A mapping μ relates elements of its domain d to elements of its range r as follows.

$$\mu(d) = r$$

A mapping resides in a data structure that supports two operations:

- `Map` links an element r in the range of μ to an arbitrary element d in the domain of μ, i.e. sets $\mu(d)$ to r.
- `Evaluate` evaluates the mapping μ for an arbitrary element d in its domain, i.e. returns $\mu(d)$.

8.2.6 Vector

For simplicity of exposition, a full vector is represented as a linear array. However, any data structure that lets you retrieve and update an arbitrary element b_i of a vector $\mathbf{b}$ based upon its index i will suffice.

8.3 Pivoting To Preserve Sparsity

As Gaussian elimination is applied to a sparse matrix $\mathbf{A}$, row operations tend to introduce nonzero elements into $\mathbf{L}$ and $\mathbf{U}$ that have no counterpart in $\mathbf{A}$. These nonzero entries in $\mathbf{L}$ and $\mathbf{U}$ that are induced by the factorization process are referred to as fill-ups. A fact central to sparse matrix techniques is that changes in the pivot strategy change the number fill-ups that occur during factorization.

This being the case, an important goal of sparse matrix algorithms is to find a pivot strategy that minimizes the number of fill-ups during LU decomposition. For the asymmetric case, Rose and Tarjan (1975) have shown that this minimization problem is NP complete. For the symmetric case, no optimal solution exists to date. Therefore, existing fill-up reduction algorithms are heuristic in nature.

8.3.1 Markowitz Pivot Strategy

Among the most successful heuristics are those based on the work of Markowitz [1957]. A Markowitz pivot strategy involves choosing a pivot element a_{ij} which minimizes a quantity called the Markowitz count at each elimination step. The Markowitz count is the product

$$(r_i - 1)(c_j - 1) \tag{91}$$

where

r_i is the number of entries in row i of the reduced matrix, and

c_j is the number of entries in column j of the reduced matrix.

Stability considerations are introduced into Markowitz pivoting strategies by defining numerical thresholds that also apply to pivot candidates. In effect, these thresholds temper sparsity considerations to preserve numerical stability. Typical threshold considerations require that the successful pivot candidate satisfy the following conditions:

$$|a_{ij}^{(k)}| \geq u \max_{l \geq k} |a_{ij}^{(k)}| \tag{92}$$

where u is a number falling in the range $0 < u \leq 1$.

Duff, Erisman, and Reid (1986) provide a thorough examination of pivoting strategies in asymmetric matrices that are based on the Markowitz criterion.

8.3.2 Minimum Degree Pivot Strategy

If a_{ji} is nonzero whenever a_{ij} is nonzero, matrix **A** has a symmetric sparsity structure. If **A** is diagonally dominant and has a symmetric sparsity structure, the minimum degree pivot strategy is commonly used to reduce fill-ups during LU decomposition. Minimum degree pivoting is a special case of Markowitz pivoting that ignores the numerical values of a_{ij} and concentrates the structure of **A**.

The most straightforward motivation of minimum degree pivoting is based on the following observations:

- Any matrix **A** with symmetric sparsity structure can be represented by an undirected, ordered graph $G = (V, E)$.
- The effect of Gaussian elimination on the sparsity structure of **A** is modeled by the impact of eliminating vertices from G.

Vertex elimination is examined elsewhere in this series of monographs–see the discussion in Section 8 (Vertex Elimination) of *Graph Algorithms* for more information. At this point, it suffices to say that a vertex v is eliminated from the graph $G = (V, E)$ by

1. Removing vertex v from $V(G)$.
2. Removing all edges that were incident upon v from $E(G)$.
3. Adding edges to $E(G)$ that connect all the vertices that were adjacent to v.

The edges that are added to G when vertex v is eliminated correspond to the fill-ups that occur in **A** when row v is eliminated.

The minimum degree pivot strategy is just the order in which the following algorithm eliminates the vertices of G.

for $i = 1, \cdots, |V|$
    Choose the vertex v from V that has the minimum degree
        Eliminate vertex v from G

You should recall that the degree of vertex v is the number of vertices that are adjacent to v. The algorithm has an arbitrary tie-breaking rule. If more than one vertex is of minimum degree at the beginning of an elimination step, any vertex from the group may be eliminated. Gomez and Franquelo (1988b), Gomez and Franquelo (1988a) examine the impact of alternate tie-breaking schemes on minimum degree pivoting strategies.

Simply put, the minimum degree algorithm pivots on the vertex which has the fewest neighbors at each elimination step. This heuristic appears to have been first described Tinney and Hart (1972). A lucid and thorough examination of the topic is found in George and Liu (1981).

8.4 Symbolic Factorization of Sparse Matrices

The goal of symbolic factorization is to define the sparsity pattern of the LU decomposition of a sparse matrix **A**. Recall that

$$\mathbf{LU} = \mathbf{PAQ} \tag{93}$$

where **P** and **Q** are row and column permutations that reflect the pivot strategy associated with the factorization process.

8.4.1 Symbolic Factorization with Minimum Degree Pivot

The goal of the current section is somewhat more specific. It describes a symbolic factorization algorithm that simulates the decomposition of **A** when a minimum degree pivot strategy is applied. The algorithm operates on an undirected graph $G = (V, E)$ whose vertices V are labeled from 1 to $|V|$. G is defined by its adjacency list A. The algorithm arrives at a symbolic factorization by eliminating vertices from G. Each stage of the process creates a reduced graph $G' = (V', E')$. The vertex v with the minimum degree in G' is chosen as the next elimination candidate. The algorithm describes the structure of **L**, **U**, **P**, and **Q** in the following manner:

- The adjacency list A is augmented to account for fill-ups that will occur during numeric factorization.
- A list L is created. The list orders the vertices in V according to a minimum degree pivot strategy. If the vertices of V are labeled according to their position in **LU**, a minimum degree ordering of V is generated.
- A mapping ψ is created. The domain of ψ is the initial label of vertex v. The range of ψ is the minimum degree label of v. That is, $\psi(v)$ is the minimum degree label of v.

Algorithm 22: Symbolic Factorization of a Sparse Matrix †

```
for i = 1, ··· , |V|
    v = minimum_degree(V′)
    while [w =iterate(A ,vertex v, 1, |V|, e)] ≠ finished
        if in_graph(V′, w)
            decrease_degree(V′, w)
            push_iteration
            while [z =iterate(A ,vertex v, w + 1, |V|, e)] ≠ finished
                e.fillup = true
                if insert(A, z, w, e)
                    increase_degree(V′, z)
                    insert(A, w, z, e)
                    increase_degree(V′, w)
            pop_iteration
    remove(V′, v)
    map(ψ, v, i)
    map(τ, i, v)
    link(L, v)
```

† *Operations and data types are defined in Section 8.2.*

- A mapping τ is created. The domain of τ is the the minimum degree label of vertex v. The range of τ is the initial label of v. That is, if $\overline{v}$ is the minimum degree label of a vertex, $\tau(\overline{v})$ is the initial label of the vertex.

One field assosciated with edge e_{ij} in the adjacency list contains a binary variable indicating whether edge (i,j) is a fill-up or not. The fill-up indicator is communicated through the buffer e in the normal manner.

Algorithm 22 computes a symbolic factorization of G that consists of these four data structures.

A few observations concerning the factorization procedure:

- The reduced graph G' is tracked in data structures designed to model vertex elimination–see Section 8.2.3 of this monograph and Section 8 (Vertex Elimination) of its companion *Graph Algorithms* for implementation details.
- The adjacency list A is augmented to log fill-ups but it is never diminished to reflect graph reduction.
- If the algorithm is efficiently implemented, ψ, τ, and l can occupy space that is vacated as G' shrinks.

8.4.2 Computational Complexity of Symbolic Factorization

The complexity of the symbolic factorization is determined by the size of the adjacency list. One iteration of the main loop requires $\frac{e^2-e}{2}$ adjacency list accesses, where e is the number of edges incident upon vertex v. If all vertices in V have the same number of incident edges, the total operation count is $|V|\frac{e^2-e}{2}$. In this case, the time complexity of symbolic factorization is $O\left(|V|\,e^2\right)$ if all operations on the data structures are $O(1)$.

Consider the following comparison of symbolic factorization operation counts to those associated with the LU decomposition of a dense matrix. If a graph has 100 vertices each of which has 5 incident edges, computing the symbolic factorization requires 1000 operations. Computing the dense factorization requires 661,650 operations. If the number of vertices increases to 1000 while the incident edges per vertex remains constant (the typical scenario in network analysis problems associated with electric power systems), symbolic factorization requires 1×10^4 operations and dense matrix decomposition requires 6×10^8 operations.

8.5 Creating $\mathbf{PAP^T}$ from a Symbolic Factorization

Symbolic factorization determines the structure of **L** and **U** when the product **PAQ** is decomposed. The current section examines a procedure for creating this product directly when **A** does not exist. Pivoting down the diagonal of the resulting matrix creates the LU decomposition predicted by symbolic factorization.

More specifically, the current section describes an algorithm which acts on the adjacency list A of an undirected graph $G = (V, E)$ to create a matrix with symmetric sparsity pattern that represents the product

$$\mathbf{PAP^T}$$

where

A is the sparse matrix that corresponds to the graph G.

P is the row permutation matrix corresponding to the minimum degree labeling of V.

It is assumed that A has been expanded to accommodate fill-ups that will occur during LU decomposition and that the permutation matrix **P** is implemented as

- A list L which traverses V in minimum degree order, and
- A mapping ψ whose domain is the initial label of a vertex v from $V(G)$ and whose range is the minimum degree label of v. That is, $\psi(v)$ returns the minimum degree label of v.

Algorithm 23: Construct $\mathbf{PAP^T}$ of a Sparse Matrix †

```
v = first(L)
for i = 1, ··· , |V|
    while [w =iterate(A ,vertex v, 1, |V|, e)] ≠ finished
        if e.fillup is true
            a = 0
        else
            make_element(A, v, w, a)
        j = ψ(w)
        insert(PAP^T, i, j, a)
    make_element(A, v, v, a)
    insert(PAP^T, i, i, a)
    a = row header
    insert(PAP^T, i, 0, a)
    v = next(L, v)
```

† *Operations and data types are defined in Section 8.2.*

Section 8.4.1 describes a procedure that creates the augmented adjacency list A and the permutation matrix $\mathbf{P}$ in the desired form.

It is further assumed that both the adjacency list A and the matrix $\mathbf{PAP^T}$ are maintained in sparse data structures that support the `scan` and `insert` operators. The operation `make_element` initializes element a_{ij} of sparse matrix $\mathbf{A}$ when its row and column indices, i and j, are specified. Communication with the data structures is maintained through buffers e and a in the normal manner.

Algorithm 23 constructs the full, asymmetric matrix $\mathbf{PAP^T}$ based on these assumptions. Zero valued entries are created for elements that will fill up during LU decomposition.

Algorithm 24 constructs the symmetric matrix $\mathbf{PAP^T}$. Zero valued entries are created for elements that will fill updduring LU decomposition.

8.6 Numeric Factorization of Sparse Matrices

Numeric factorization algorithms work with nonzero values of a sparse matrix $\mathbf{A}$ and the data structures resulting from symbolic factorization to compute the factorization

$$\mathbf{LU} = \mathbf{PAP^T}$$

Algorithms discussed in the current section act on a sparse $n \times n$ matrix $\mathbf{A}$. They assume that

Algorithm 24: Construct $\mathbf{PAP^T}$ of a Sparse Symmetric Matrix †

```
v = first(L)
for i = 1, ··· , |V|
    while [w = iterate(A, vertex v, 1, |V|, e)] ≠ finished
        j = ψ(w)
        if j > i
            if e.fillup is true
                a = 0
            else
                make_element(A, v, w, a)
            insert(PAP^T, i, j, a)
    make_element(A, v, v, a)
    insert(PAP^T, i, i, a)
    v = next(L, v)
```

† *Operations and data types are defined in Section 8.2.*

- **A** already reflects the pivot strategy defined by **P** and $\mathbf{P^T}$, i.e. the algorithms pivot down the diagonal.
- **A** has zero-valued entries at fill-up locations.
- **A** is maintained in a sparse data structure supporting the `get`, `scan`, and `put` operators. Communication with the data structure is maintained through the buffer a in the normal manner.

Techniques for creating the requisite pivot ordered, fill-up augmented **A** matrix (i.e. $\mathbf{PAP^T}$) are discussed in Section 8.5.

Algorithm 25 uses Doolittle's method to compute the LU decomposition of a sparse matrix **A**.The algorithm overwrites **A** with **LU**.

Algorithm 26 uses Doolittle's method to compute **U**, the upper triangular factors, of a symmetric sparse matrix **A**. It is assumed that **A** is initially stored as an upper triangular matrix. The algorithm overwrites **A** with **U**. The vector **w** is used to construct the nonzero entries of each column from **U**. The vector **c** contains cursors to the row in **L** with which the entries of **A** are associated, e.g. if w_k contains l_{ji} then c_k is j.

Doolittle's method for full, asymmetric matrices, is examined in Section 4.2. Doolittle's method for full, symmetric matrices is discussed in Section 7.4.

Algorithm 25: LU Decomposition of a Sparse Matrix by Doolittle's Method †

```
for i = 2, ··· , n
    while [ j =scan (A ,row i, 1, n, a)] ≠ finished
        push_scan
        α = a
        m = min (i − 1, j − 1)
        while [ p =scan (A ,row i, 1, m, a)] ≠ finished
            b = a
            if get (A, p, j, a) is successful
                α = α − ab
        if i > j
            get (A, j, j, a)
            α = α / a
        a = α
        put (A, i, j, a)
        pop_scan
```

† *Operations and data types are defined in Section 8.2.*

Algorithm 26: LU Decomposition of Sparse Symmetric Matrix by Doolittle's Method †

```
for i = 1, ··· , n
    k = 0
    for j = 1, ··· , i − 1
        if get (A, j, i, a) is successful
            k = k + 1
            w_k = a
            c_k = j
            get (A, j, j, a)
            w_k = w_k / a
    while [ j =scan (A ,row i, i, n, a)] ≠ finished
        α = a
        for p = 1, ··· , k
            if get (A, c_p, j, a) is successful
                α = α − a w_p
        a = α
        put (A, i, j, a)
```

† *Operations and data types are defined in Section 8.2.*

Algorithm 27: Permute **b** to order **P**

for $i = 1, \cdots, n$
    $k = \psi(i)$
    $y_i = b_k$

8.7 Solving Sparse Linear Systems

As we have seen, the LU decomposition of the matrix **PAQ** is used to solve the linear system of equations **Ax** = **b** by sequentially solving Equation 58 through Equation 61, which are repeated below.

$$\begin{aligned} \mathbf{y} &= \mathbf{Pb} \\ \mathbf{Lc} &= \mathbf{y} \\ \mathbf{Uz} &= \mathbf{c} \\ \mathbf{x} &= \mathbf{Qz} \end{aligned}$$

Sparsity techniques benefit this process. The algorithms presented in this section assume:

- An LU decomposition of **PAQ** exists.
- The factorization is maintained in a sparse matrix data structure which supports the `scan` and `get` operators. These operators are described in Section 8.2.1.
- The row permutations **P** are represented by a mapping ψ whose domain is a row index in **A** and whose range is the corresponding row index in **PAQ**.
- The column permutations **Q** are represented by a mapping τ whose domain is a column index in **PAQ** and whose range is the corresponding column index in **A**.

For example, Section 8.6 describes algorithms that create numeric factorizations satisfying the first two of these assumptions. Section 8.4.1 describes an algorithm for obtaining the row and column permutations corresponding to a minimum degree pivot strategy.

For simplicity of exposition, it is assumed that the vectors **b**, **c**, **x**, **y**, and **z** are stored in linear arrays. However, any data structure that lets you retrieve and update an element of a vector based on its index will suffice.

8.7.1 Permute the Constant Vector

The equation **y** = **Pb** is efficiently implemented using the mapping ψ to permute the elements of **b**. Algorithm 27 illustrates this procedure.

Algorithm 28: Sparse Forward Substitution

for $i = 1, \cdots, n$
  $\alpha = y_i$
  while $[j = \texttt{scan}(\mathbf{L}, \text{row } i, 1, i-1, l)] \neq$ finished
    $\alpha = \alpha - l y_i$
  $\texttt{get}(\mathbf{L}, i, i, l)$
  $c_i = \dfrac{\alpha}{l}$

Algorithm 29: Symmetric Sparse Forward Substitution

for $i = n, \cdots, 1$
  $c_i = y_i$
  $\texttt{get}(\mathbf{U}, i, i, u)$
  $\alpha = \dfrac{c_i}{u}$
  while $[k = \texttt{scan}(\mathbf{U}, \text{row } i, i+1, n, u)] \neq$ finished
    $y_k = y_k - u\alpha$

8.7.2 Sparse Forward Substitution

The lower triangular system of equations $\mathbf{Lc} = \mathbf{y}$ is solved by forward substitution. Algorithm 28 implements the inner product formulation of forward substitution on a sparse $\mathbf{L}$ and full vectors $\mathbf{c}$ and $\mathbf{y}$.

The sequencing of the operations permits the use of a single vector $\mathbf{y}$. Operations that update c_i would overwrite y_i instead. If $\mathbf{L}$ is unit lower triangular, division by the diagonal element l_{ii} is omitted.

Algorithm 29 implements an outer product formulation of forward substitution for use with symmetric systems whose upper triangular factors are available. See Section 5.3 and Section 7.6 for additional information.

8.7.3 Sparse Backward Substitution

The upper triangular system of equations $\mathbf{Uz} = \mathbf{c}$ is solved by backward substitution. Algorithm 30 implements the inner product formulation of backward substitution on a sparse $\mathbf{U}$ and full vectors $\mathbf{c}$ and $\mathbf{z}$.

The sequencing of the operations permits the use of a single vector $\mathbf{c}$. Operations that update z_i would overwrite c_i instead. If $\mathbf{U}$ is unit upper triangular, division by the diagonal element u_{ii} is omitted.

Algorithm 30: Sparse Back Substitution

for $i = n, \cdots, 1$
  $\alpha = c_i$
  while $[j = \texttt{scan}(\mathbf{U}, \text{row } i, i+1, n, u)] \neq \text{finished}$
    $\alpha = \alpha - uz_j$
  $\texttt{get}(\mathbf{U}, i, i, u)$
  $z_i = \dfrac{\alpha}{u}$

Algorithm 31: Permute **x** to order **Q**

for $i = 1, \cdots, n$
  $k = \tau(i)$
  $x_i = z_k$

8.7.4 Permute the Solution Vector

The equation $\mathbf{x} = \mathbf{Qz}$ is efficiently implemented using the mapping τ to permute the elements of $\mathbf{z}$. Algorithm 31 illustrates this process.

8.8 Sparse LU Factor Update

If the LU decomposition of the sparse product **PAQ** exists and the factorization of a related matrix

$$\mathbf{PA'Q} = \mathbf{P}(\mathbf{A} + \mathbf{\Delta A})\mathbf{Q} \tag{94}$$

is desired, factor update is at times the procedure of choice. The current section examines procedures for updating the sparse factorization of **PAQ** following a rank one modification of **A**, that is

$$\mathbf{A'} = \mathbf{A} + \alpha \mathbf{y}\mathbf{z}^{\mathrm{T}} \tag{95}$$

where α is a scalar, **y** and **z** are n vectors, and $\mathbf{A}'$ has the same sparsity pattern as **A**.

The condition on the structure of $\mathbf{A}'$ is not imposed by the factor update process, but is instead a comment on the utility of factor update in a sparse environment. If the modification to **A** introduces new elements into the matrix, the pivot sequence determined during symbolic factorization may no longer apply. The sparsity degradation introduced by an inappropriate pivot sequence may outweigh the benefits gained from the updating the existing factorization.

The performance of factor update algorithms is often enhanced by restricting pivot operations to the portions of **L** and **U** that are directly effected by the change in **A**. Papers

Algorithm 32: Factorization Path

while $i \neq$ finished
$j =$ `scan` (**U**, row $i, i+1, n, u$)
$k =$ `scan` (**L**, column $i, i+1, n, l$)
$i =$ `min` (j, k)

Algorithm 33: Symmetric Factorization Path

while $i \neq$ finished
$i =$ `scan` (**U**, row $i, i+1, n, u$)

by Tinney, Brandwajn, and Chan (1985) and Chan and Brandwajn (1986) describe a systematic methodology for determining this subset of **LU**. The rows of **U** and columns in **L** that are changed during factor update are referred to as the factorization path. The fundamental operation is to determine the factorization path associated with a vector **y** with just one nonzero element. Such a vector is called a singleton. Its factorization path is called a singleton path. If more than one of the elements in **y** are nonzero, the composite factorization path is simply the union of the singleton paths.

8.8.1 Factorization Path of a Singleton Update

If the LU decomposition of **PAQ** is stored as an asymmetric sparse matrix and a singleton vector **y** has one nonzero element at location i, its factorization path through **LU** is determined by Algorithm 32. Each value ascribed to i by Algorithm 32 is a vertex on the factorization path of **y**.

In words, Algorithm 32 starts at u_{ii} (the diagonal element in **U** corresponding to the nonzero element in **y**) and looks to the right until it finds the first nonzero element u_{ij}. It then starts at element l_{ii} and looks down column i of **L** until it finds a nonzero element l_{ik}. The value of i is then reset to the smaller of j or k and the process is repeated for the next u_{ii} and l_{ii}. The procedure ends when there are no elements to the right of the diagonal in **U** or below the diagonal in **L** for some vertex on the factorization path. Obviously, it is assumes that a column scan is independent of a row scan but works in a similar manner.

If the LU decomposition of $\mathbf{PAP^T}$ has a symmetric sparsity pattern Algorithm 32 simplifies to Algorithm 33.

The symmetry makes the search through column i of **L** unnecessary. In Algorithm 33, the use of **U** to determine the factorization path was arbitrary. Either **L** or **U** can anchor the process.

8.8.2 Revising LU after a Singleton Update

Algorithm 34 updates a structurally symmetric LU factorization after a rank one modification to **A**. It assumes:

- **L** is unit lower triangular and maintained in a sparse data structure that communicates using the buffer *l*.
- **U** is upper triangular and maintained in a sparse data structure that communicates using the buffer *u*.
- The sparse data structure does not permit a column scan.
- The **y** vector is full and all entries are zero except y_i.
- The **z** vector is full.
- The product $\mathbf{y}\mathbf{z}^{\mathbf{T}}$ has the same sparsity structure as **A**.
- The **y** and $\mathbf{z}^{\mathbf{T}}$ vectors have been permuted by **P** and $\mathbf{P}^{\mathbf{T}}$ so they are in the same frame of reference as **L** and **U**.

The requirement that **L** and **U** occupy different data structures in Algorithm 34 is pedantic. In practice, **L** will occupy the subdiagonal portion of a sparse matrix and **U** will occupy the diagonal and superdiagonal portions of the same matrix.

If **A** is symmetric, $\mathbf{y} = \mathbf{z}$, and $\mathbf{A}'$ has the same sparsity pattern as **A**, Algorithm 34 simplifies to Algorithm 35.

9 Implementation Notes

This document concludes with a brief discussion of an experimental implementation of sparse matrix algorithms in a highly cached database environment. It was written as part of the documentation of an experimental project, from a bygone era, which proceeded along these lines.

The relevance of the material in his section to current readers is debatable since it examines software implementations for machine architectures that are no longer in use (or available). Nonetheless, some aspects of the discussion are still pertinent despite changes in the execution environment. For this reason and for the sake of completeness, it was decided to retain the information in this monograph. Do with it as you like.

9.1 Sparse Matrix Representation

Data structures used to maintain sparse matrices must provide access to the nonzero elements of a matrix in a manner which facilitates efficient implementation of the algorithms that are examined in Section 8. The current sparse matrix implementation also seeks to support a high degree of generality both in problem size and the definition of

Algorithm 34: Structurally Symmetric Sparse LU Factor Update

while $i \neq$ finished
  `get` $(\mathbf{U}, i, i, u)$
  $\delta = u$
  $p = y_i$
  $q = z_i$
  $u = u + \alpha pq$
  `put` $(\mathbf{U}, i, i, u)$
  $\beta_1 = \dfrac{\alpha p}{u}$
  $\beta_2 = \alpha q$
  $q = \dfrac{q}{\delta}$
  $\delta = \dfrac{u}{\delta}$
  $\alpha = \dfrac{\alpha}{\delta}$
  $k =$ finished
  while $[j =$`scan` $(\mathbf{U}, \text{row } i, i+1, n, u)] \neq$ finished
    if j is first element on row i
      $k = j$
    $z_j = z_j - qu$
    $u = \delta u + \beta_2 z_j$
    `put` $(\mathbf{U}, i, j, u)$
    `get` $(\mathbf{L}, j, i, l)$
    $y_j = y_j - pl$
    $l = l + \beta_1 y_j$
    `put` $(\mathbf{L}, j, i, l)$
  $i = k$

a matrix element. Among other things, this implies that the algorithms must be able to solve problems that are too large to fit into core. A B^{link} tree supported by a data base cache (described elsewhere, see also Lehman and Yao (1981)) is one possible vehicle for this task. Simply put, the fundamental sparse matrix data structure is:

- Each matrix is a relation in a data base, and
- Each nonzero element of a matrix is a tuple in a matrix relation.

Matrix tuples have the structure indicated in Figure 4.

The row and column domains of each tuple constitute a compound key to the matrix relation. Their meaning corresponds to the standard dense matrix terminology.

The description of a matrix element is left intentionally vague. Its definition varies with the application. Matrix elements must include a real number, double precision real

Algorithm 35: Symmetric Sparse LU Factor Update

while $i \neq$ finished
  `get` $(\mathbf{U}, i, i, u)$
  $\delta = u$
  $p = y_i$
  $u = u + \alpha p^2$
  `put` $(\mathbf{U}, i, i, u)$
  $\beta = \dfrac{\alpha p}{u}$
  $q = \dfrac{q}{\delta}$
  $\delta = \dfrac{u}{\delta}$
  $\alpha = \dfrac{\alpha}{\delta}$
  $k =$ finished
  while $[j =$`scan` $(\mathbf{U}$,row $i, i+1, n, u)] \neq$ finished
    if j is first element on row i
      $k = j$
    $y_j = y_j - pu$
    $u = \delta u + \beta z_j$
    `put` $(\mathbf{U}, i, j, u)$
  $i = k$

Figure 4: Matrix Tuple Structure

number, complex number, or any other entity for which the arithmetical operations of addition, subtraction, multiplication, and division are reasonably defined.

In this context, matrix elements are accessed through high level data base operations:

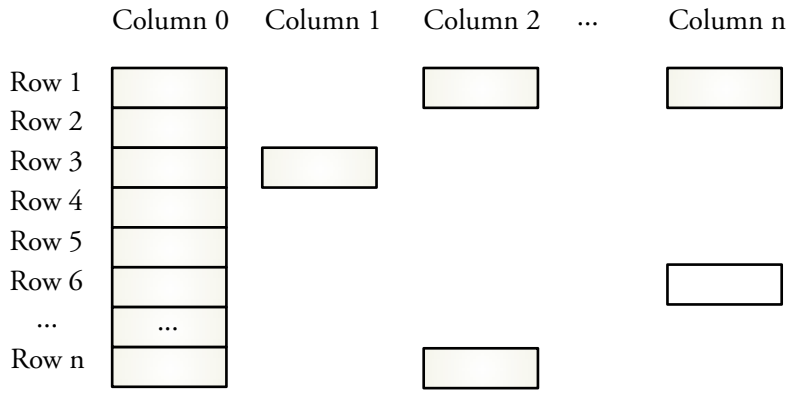
- `Get` retrieves a random tuple.
- `Next` retrieves tuples sequentially. You will recall that the scan operator (defined in Section 8.2.1 and Section 8.2.2) is used extensively by sparse matrix algorithms in Section 8. Scan is implemented by embellishing the `next` primitive.
- `Put` updates the non-key portions of an existing tuple.
- `Insert` adds a new tuple to a relation.
- `Delete` removes an existing tuple from a relation.

This data structure places few constraints on the representation of a matrix. However, several conventions are adopted to facilitate consistent algorithms and efficient cache access:

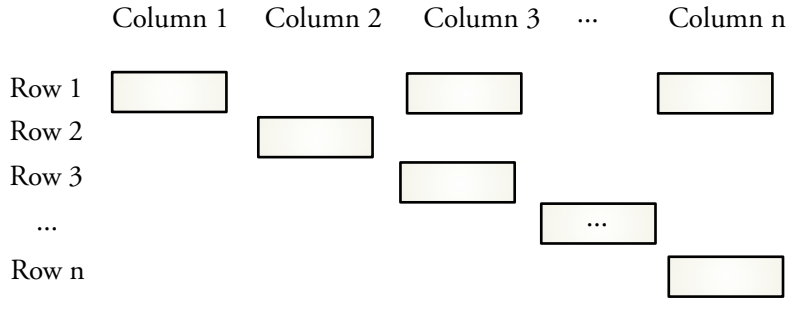
- Matrices have one-based indexing, i.e. the row and column indices of an $n \times n$ matrix range from 1 to n.
- Column zero exists for each row of an asymmetric matrix. Column zero serves as a row header and facilitates row operations. It does not enter into the calculations.
- A symmetric matrix matrix is stored as an upper triangular matrix. In this representation, the diagonal element anchors row operations as well as entering into the computations. Column zero is not used for symmetric matrices.

Figure 5 depicts the data structure associated with sparse matrices.

Figure 5: Sparse Matrix Representation



Asymmetric Matrix



Symmetric Matrix

9.2 Database Cache Performance

When matrix algorithms are implemented in a relational environment, database access requirements can play a significant (if not dominating) role in an algorithm's time complexity. The current section examines the theoretical and empirical performance characteristics of a B^{link} tree (see Lehman and Yao (1981)) supported by an in-core cache with a LRU paging discipline. The operators described in the previous section form the basis of the discussion.

9.2.1 Sequential Matrix Element Retrieval

The time complexity of `next` is O(1), since `next` just looks up a virtual address (no search is involved). A `next` operation requires one cache access. No key comparisons are necessary. An additional cache access is required to obtain the non-key portion of the tuple.

9.2.2 Arbitrary Matrix Element Retrieval

The time complexity of `get` is O($\log n$) where n is the number of tuples in the relation. A `get` operation is implemented by a current tree search. When the key to a sparse matrix is four bytes long, the current tree of the matrix is a 31–61 tree. Interpolating the tables found in Gonnet (1984) yields:

- The expected height of a 10,000 key current tree (i.e. number of nodes searched) is three.
- The average node to key ratio of a 10,000 key current tree is 0.02904. Hence, the average node will contain 34.43 keys.

Each descending branch in a current tree search is determined through a binary search of a tree node. The maximum number of key comparisons needed to search a node in this manner is $\log_2 k$, where k is the number of keys in the node. Therefore, it will take no more than 5.1 comparisons to locate the appropriate tree branch in an average 35 key node.

These observations imply that no more than 3 cache lookups and 16 key comparisons are required to locate an entry in a 10,000 key current tree. An additional cache access is required to obtain the non-key portions of the tuple.

9.2.3 Arbitrary Matrix Element Update

If the address of the non-key portion of a tuple is known, `put` is an O(1) operation requiring a single cache access. If the address is not known `put` is equivalent to a `get` – O($\log n$) – followed by a direct cache access.

9.2.4 Matrix Element Insertion

The time complexity of `insert` is O(log n) where n is the number of tuples in the relation. An `insert` operation is equivalent to a `put` operation unless the tree splits. Interpolating the tables in Gonnet (1984) yields:

- The average number of splits for the $n{+}1^{st}$ insertion into a 10,000 key 31–61 tree is approximately 0.02933, i.e. the tree will split each time 33.40 items are inserted (on the average).

Splitting increases the constant associated with the growth rate slightly. It does not increase the growth rate *per se*.

9.2.5 Matrix Element Deletion

Deleting a key from a B^{link} tree is analogous to inserting one, except tree nodes occasionally combine instead of splitting. Therefore, the time complexity of delete is $O(\log n)$ like insertion.

9.2.6 Empirical Performance Measurements

The measurements in Table 1 provide empirical performance statistics for various data base operations. The measurements were made on a 16 Mhz IBM PS/2 Model 70 with a 16 Mhz 80387 coprocessor and a 27 msec, 60 Mbyte fixed disk drive (do you think the hardware is a bit dated? I suspect many readers have only seen this hardware in a museum — if at all). You should note two characteristics of the measurements:

- The cache was large enough to hold the entire B^{link} tree of relations A, B, and C. There are no cache faults to disk in these measurements. The relation D was too big to fit in core. Its `next` times reflect numerous cache faults.
- The `get` operation looked for the same item during each repetition. This is explains the lack of cache faults while relation D was processed. Once the path to the item was in core it was never paged out.

Neglecting cache faults, the time required to find a tuple is a function of two variables: the size of the relation and the size of the key. The number of tuples in a relation determines the number of comparisons that are made. The size of the key effects the amount of work required to perform each comparison. Comparing the "get relation D" measurements of Table 1 to the "get relation C" measurements provides an indication of the actual effect of a relation's size on search time (since both relations have the same size key). Note that

$$\frac{2,520\ \mu\text{sec}}{2,165\ \mu\text{sec}} \approx 1.167$$

Table 1: Database Cache Benchmarks

Operation	Repetitions 30k (seconds)	50k (seconds)	100k (seconds)	200k (seconds)	Average (μsec)
Relation A [1]					
Next	n/a	12	25	51	255
Get	n/a	26	52	103	515
Relation B [2]					
Next	7	12	24	47	235
Get	34	57	114	228	1,140
Relation C [3]					
Next	7	12	24	49	245
Get	65	108	216	433	2,165
Relation D [4]					
Next	48	82	164	n/a	1,640
Cache faults	2,058	3,541	7,095		
Get	76	126	252	n/a	2,520
Cache faults	1	1	1		

[1] 112 tuples, 2 byte key.
[2] 463 tuples, 4 byte key.
[3] 673 tuples, 22 byte key.
[4] 10,122 tuples, 22 byte key.

which is below the theoretical bound

$$\frac{\log_2 10,122}{\log_2 673} \approx \frac{13.31}{9.40} \approx 1.416$$

Comparing "get relation C" to "get relation B" gives a good feel for the impact of key size. The size of the relation should not have much impact on the search time discrepancies since

$$\frac{\log_2 673}{\log_2 463} \approx \frac{9.40}{8.87} \approx 1.060$$

The `next` operation was metered by repeatedly scanning a relation until the desired number of operations was achieved. Each time the end of a relation was encountered the scan was restarted. The high `next` measurement for relation A probably reflects the overhead of many loop starts and stops (since there were only 12 tuples in the relation). The elevated `next` time of the relation C is probably due to key length. After all the keys on a cache page are processed, a relatively expensive page fault occurs. Larger

keys cause the cache page to change more frequently. Given that the in-core `next` observations have a mean of 240 μsec and a standard deviation of 10 μsec, the effects of these peripheral processes appear to be relatively insignificant.

In summary, Table 1 shows that the O(1) operations have an empirical time advantage that ranges from 1.54/1 to 8.84/1 over the O(log n) operations. This observation underscores the importance of tailoring algorithmic implementations to take advantage of the O(1) operators. In practical terms, his means that sequential access and stack operations are preferred to direct random access.

9.3 Floating Point Performance

Note. *This section contains data that is so antiquated that we're not sure it has much relevance to modern hardware configurations. It is included for the sake of completeness and as a historical curiosity.*

Table 2: Floating Point Benchmarks

	With 80387		No 80387	
Operation	Repetitions 2,000k (seconds)	Average Time (μsec)	Repetitions 200k (seconds)	Average Time (μsec)
Add	25.9	13.0	34.3	172
Subtract	25.9	13.0	35.3	177
Multiply	28.4	14.2	44.3	222
Divide	33.6	16.8	50.9	255
Inner product [1]	40.3	20.2	61.9	310
Scalar multiply [2]	30.2	15.1	45.0	225
Loop overhead		1.3		1.3

[1] sum + = a[cursor[i]] * y[cursor[j]]
[2] a[cursor[i]] * = scalar

A variety of floating point operations were monitored under MS DOS version 3.30 on a 16 Mhz IBM PS/2 Model 70 with a 16 Mhz 80387 coprocessor and a 27 msec, 60 Mbyte fixed disk drive. The 32 bit operations available on the 80386 were not used. Table 2 catalogs the time requirements of the simple arithmetical operations, inner product accumulation, and multiplying a vector by a scalar. All benchmarks were performed using double precision real numbers. The test contains a loop that was restarted after every 500 operations, e.g. 200k repetitions also includes the overhead of starting and

stopping a loop 400 times. With this testing scheme, all loop counters and array indices were maintained in the registers.

The measurements in Table 3 provide a similar analysis of math functions in the Microsoft C Version 5.1 math library. These benchmarks were conducted with a single loop whose counter was a long integer.

Table 3: Math Library Benchmarks

	With 80387		No 80387	
	Repetitions 300k	Average Time	Repetitions 10k	Average Time
Function	(seconds)	(μsec)	(seconds)	(μsec)
acos	36.2	121	30.5	3,050
asin	35.1	117	29.9	2,990
atan	26.0	87	23.0	2,300
cos	37.7	126	25.3	2,530
sin	37.0	123	24.7	2,470
tan	31.7	106	19.2	1,920
log	25.4	85	18.5	1,850
sqrt	16.5	55	5.7	570
pow	51.4	171	38.6	3,860
j0 [1]	235.1	784	60.7	6,070
j6	662.0[2]	2,207	176.3	17,603
y0 [3]	510.0[2]	1,700	146.4	14,640
Loop overhead		3		3

[1] Bessel function of the first kind, order 0.
[2] Extrapolated from 30,000 repetitions.
[3] Bessel function of the second kind, order 0.

Differences in loop overheads found in Table 2 and Table 3 are accounted for by the differences in the loop counter implementation described above. The 3 μsec overhead reflects the time required to increment a long integer and monitor the termination condition (which also involved a long integer comparison). The 1.3 μsec overhead reflects the time required to increment a register and monitor the termination condition (which involved a register comparison).

9.4 Auxiliary Store

A data structure referred to as the auxiliary store is provided to support temporary information that is required by matrix algorithms but not stored in the matrix relations

themselves. The auxiliary store is implemented in a manner that takes advantage of unused heap space at execution time. If the heap is big enough to accommodate the entire auxiliary store, an array of structures is allocated and the store is maintained in core. If available heap space is inadequate, a relation is allocated and the auxiliary store is maintained in the database cache. Access to the in-core version of the auxiliary store requires 13.8 μsec. Heap access time does not vary with the size of the store.

References

Bennett, J. (1965). "Triangular Factors of Modified Matrices". In: *Numerische Mathematik* 7, pp. 217–221 (cit. on p. 32).

Chan, S. and V. Brandwajn (1986). "Partial matrix refactorization". In: *IEEE Transactions on Power Systems* 1.1, pp. 193–200 (cit. on pp. 32, 58).

Conte, S. and C. de Boor (1972). *Elementary Numerical Analysis*. New York: McGraw-Hill Book Company (cit. on pp. 20, 24).

Duff, I. S., A. M. Erisman, and J. K. Reid (1986). *Direct Methods for Sparse Matrices*. Oxford: Clarendon Press (cit. on pp. 17, 20, 43, 48).

Fox, L. (1964). *An Introduction to Numerical Linear Algebra*. Oxford: Clarendon Press (cit. on p. 17).

George, Alan and Joseph W.H. Liu (1981). *Computer Solutions of Large Sparse Positive Definite Systems*. Engle Wood Cliffs, New Jersey: Prentice-Hall. ISBN: 9780131652743 (cit. on pp. 29, 49).

Gill, P. et al. (1974). "Methods for Modifying Matrix Factorizations". In: *Mathematics of Computation* 28.126, pp. 505–535 (cit. on pp. 32, 40).

Golub, Gene H. and Charles F. van Van Loan (1983). *Matrix Computations*. Baltimore: Johns Hopkins University Press (cit. on pp. 17, 18, 24).

Gomez, A. and L. Franquelo (1988a). "An efficient ordering algorithm to improve sparse vector methods". In: *IEEE Transactions on Power Systems* 3.4, pp. 1538–1544 (cit. on p. 49).

— (1988b). "Node ordering algorithms for sparse vector method improvement". In: *IEEE Transactions on Power Systems* 3.1 (cit. on p. 49).

Gonnet, G. (1984). *Handbook of Algorithms and Data Structures*. Reading, Massachusetts: Addison-Wesley (cit. on pp. 63, 64).

Hager, W. (1989). "Updating the Inverse of A Matrix". In: *SIAM Review* 31.2, pp. 221–239 (cit. on p. 32).

Lehman, P. and B. Yao (1981). "Efficient Locking for Concurrent Operations on B-Trees". In: *ACM Transaction on Database Systems* 6.4, pp. 650–669 (cit. on pp. 60, 63).

Press, W. H. et al. (1988). *Numerical Recipes in C*. Cambridge and New York: Cambridge University Press. ISBN: 9780521354660 (cit. on pp. 17, 24).

Rose, D. and R. Tarjan (1975). "Algorithmic aspects of vertex elimination". In: pp. 245–254 (cit. on p. 47).

Tinney, W., V. Brandwajn, and S. Chan (1985). "Sparse vector methods". In: *IEEE Transactions on Power Apparatus and Systems* 104.2 (cit. on pp. 29, 58).

Tinney, W. and C. Hart (1972). "Power flow solution by Newton's method". In: *IEEE Transactions on Power Apparatus and Systems* 86.6 (cit. on pp. 42, 43, 49).

Tinnney, W. and J. Walker (1967). "Direct solutions of sparse network equations by optimally ordered triangular factorization". In: *Proceedings of the IEEE* 55.11, pp. 1801–1809 (cit. on p. 22).