Implementing Matrix Computation Algorithms

© Mark H. Bishop, 2007
mb@mark-bishop.net

Cite this reference as: *Implementing Matrix Computation Algorithms*, (2007), Mark Bishop, New England Testing Laboratory, Providence, RI.

DISCLAIMER

Preface

The target audience for this book is the community of students, scientists, engineers and social scientists who have an interest in applying matrix computations to solve problems by writing custom applications. The applications themselves are myriad and include (but are by no means limited to) signal processing, differential equations, image processing, simultaneous equations, and linear regression. The typical reader will have a primary interest in a specific application, and a resultant interest in learning — at a functional level — the mathematics of matrix computations. The reader should have strong mathematical and logical skills, but advanced coursework is not a prerequisite. The reader's background should include a familiarity with one or more programming languages. Programmers can easily adapt the C# code used in this book to other languages.

There are a few very good texts that treat the mathematics of matrix computations. These books provide algorithms, develop mathematical proofs, and discuss at length the necessary techniques (and thinking) required to develop efficient, reliable implementations. This book focuses on the implementation of established algorithms using the C# programming language. It also provides a proof-free development of concepts needed to understand the implementations, as well as several worked examples. It is my hope that the reader can use the material provided herein and be "up and running" with custom implementations in short order.

I recommend augmenting this book with a text on matrix computation mathematics. One very complete and plainly presented example is *Matrix Computations 3rd Edition* by Gene H. Golub and Charles F. Van Loan (Johns Hopkins University Press, Baltimore, 1996). I have relied heavily on this resource in writing this book.

Contents

1. Introduction

An algorithm is a step-by-step procedure for solving a problem or performing a task. With matrix computations, the term is most often applied to a mathematical idea. Algorithms are often presented in a symbolic language called *pseudo-code*, but they can also be stated in text form. An implementation of an algorithm is a working expression of the mathematical idea in a computer language. The algorithm is a construct of mathematical science. The implementation is an algorithm's encapsulation for computer technology.

This book is about implementations. We will explore implementing algorithms for matrix computations and obtaining solutions to linear systems. You will want to arm yourself with a good reference for matrix algorithms (see the preface for one such source). We will qualitatively address some underlying mathematics for the computations, then we will present the implementation. To fully benefit from this approach you must read through the implementations and determine exactly what is happening. Then, you should mercilessly criticize the implementation and write a better one. That is how to learn this subject.

The approach for this book is to write a matrix computations class library, which can be built into a dynamic link library (DLL) for inclusion in any application/interface. First, however, we need some methods.

Conventions and General Background

Basic Definitions

Volumes can be written about number theory. For utilitarian purposes, the following concepts provide a working model.

Real Numbers:

If we draw a line from negative infinity to positive infinity, then any value that can be represented by a point on that line is a real number. We denote the set of all real numbers: □ .

Imaginary numbers:

Imaginary numbers are numbers that are a real number multiple of $i = \sqrt{-1}$, e.g. $2i$ or $-0.235i$.

Complex numbers:

Complex numbers have a real component and an imaginary component. For example, the complex number $3 + 4i$ has a real component 3 and an imaginary

component $4i$. It has a magnitude and an unusual sort of direction. We define the complex conjugate of the complex number c=a+bi as c*=a – bi.

Complex numbers use the following basic arithmetic rules (see the Wolfram Research website):

Addition/Subtraction: $(a+bi)\pm(c+di)=(a\pm c)+(b\pm d)i$

Multiplication: $(a+bi)(c+di)=(ac-bd)+(ad+bc)i$

Division: $\dfrac{(a+bi)}{(c+di)}=\dfrac{(ac+bd)+(bc-ad)}{c^2+d^2}$

Scalars:

Traditionally, a scalar is a quantity that has magnitude but no direction. Practically, a scalar is a single valued number.

Vectors:

Vectors are ordered sets of mathematical elements. In many applications they are understood to have magnitude and direction. More abstractly, they have magnitude and dimensional multiplicity.

Matrices:

A matrix is an array of mathematical elements having horizontal rows and vertical columns. In this book, the number of rows is designated m and the number of columns is n. If we restrict our consideration to real numbers, then we can represent the set of all m-by-n matrices as: $A\in\Re^{mxn}$. For complex systems we may designate the corresponding set as $A\in\Re^{mxn}$. We designate a matrix with a capital letter. The elements of the matrix are designated with appropriately subscripted lowercase letters:

$$A\in\Re^{mxn}=\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

The element a_{ij} resides in the i^{th} row and the j^{th} column of the matrix.

n -tuples:

An *n*-tuple is an ordered set of n elements. It is more general in sense than a vector, but in relation to the subject matter at hand, we can use *n*–tuple and n-dimensional vector interchangeably.

The terms *point*, *line*, *plane* and *space* are not rigorously defined. Our intuition about these concepts works much better than the circular arguments that arise when we try to constrain their meaning to definitions. As for extension beyond three (or four) dimensions, any definitions are mostly philosophical.

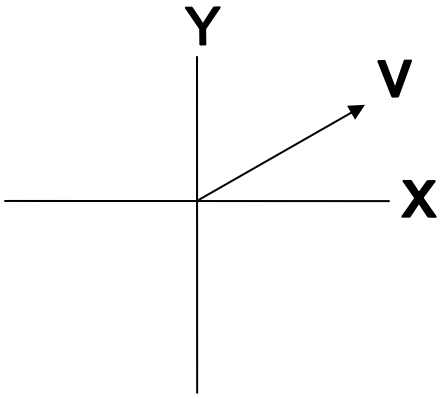
Introductory Vector and Matrix Terminology

Our discussion here is not intended to be a survey of the language of linear algebra, but instead will be limited to ideas that have immediate relevance to the subject matter discussed later in this book.

Vectors may be effectively conceptualized as special case matrices where the number of columns is one (column vectors) or the number of rows is one (row vectors). The set of all n dimensional vectors make up an n dimensional *vectorspace*. A subset of those vectors is termed a *subspace*. The set of all linear combinations of vectors in a vector subspace is referred to as the *span* of those vectors. It is important to recognize that real world vectors may have referent objects such as forces or displacements. A force in three dimensional space does not occupy the same vector space as a displacement in three dimensional space. Physical and kinematic objects are frequently represented by vectors. Such vectors have intuitively understandable magnitude and direction. However, there is an indefinite number of pure and applied mathematical systems — such as n^{th} order polynomials — which may be represented by vectors.

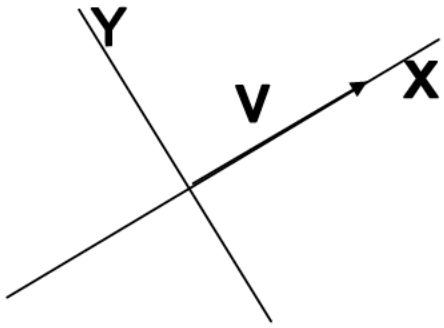
Basis

Representing vectors and managing vector computations most often require us to resolve the vectors into spatial components. For systems that literally exist in two or three dimensional space, this frequently involves the familiar Cartesian coordinate system. This system is a construction of two or three mutually perpendicular axes that intersect at the same point, which we designate (0,0) or (0,0,0). Each of these axes has a unit vector (a vector of length one) associated with it. Most texts reserve the variable e to represent these unit vectors. Consider the following two dimensional example:



The vector v has the length and direction shown. It has these properties *without* reference to the coordinate system. We can represent the vector in terms of its Cartesian

coordinates as $v = 3e_x + 2e_y$. The vector is fixed but there is nothing sacred about our coordinate system. The system below is equally adequate



where $v = 3.6e_x + 0e_y$. The vector is still exactly as long as it was and points directly where it used to. In fact, any two nonparallel vectors g_1 and g_2 can be used to spatially describe v. They need not be of unit length and they do not need to be perpendicular. When used in this fashion, g_1 and g_2 are called *base vectors*. The set of base vectors is collectively referred to as a *basis*. For an n-dimensional space, e_n represents a Euclidian basis and is given by the following set of base vectors:

$e_n = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$, where the vectors are either the rows or columns of the matrix.

Similarly, a generalized basis may be represented by the matrix $G_n = \{g_1, g_2, \ldots g_n\}$, where $g_1, g_2, \ldots g_n$ are the base (column) vectors (frequently given in Cartesian components). We have seen that for two dimensions, the requirement for a basis is that the vectors be nonparallel. In three dimensions, the vectors must be nonparallel and they must not be all in one plane. In n-dimensions, G_n is a basis if and only if it has n *linearly independent* n-dimensional vectors.

Linear Independence

A set of i vectors is said to be linearly independent when $0 = \sum_{n=1}^{i} \alpha_n v_n$ only has a solution when all of the constants α_n are zero.

The Vector Dot Product

The dot product (or inner product) of two m dimensional vectors v and u is $< v,u >= \sum_{i=1}^{m} v_i u_i$. The result is a scalar. If the dot product of two vectors is zero, then and only then are the vectors said to be orthogonal. Orthogonal vector pairs can intuitively be thought of as perpendicular.

Other Common Terms

The Euclidian vector norm is defined as: $\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}$. The Euclidian norm is interpretable as the vector's length or magnitude.

Elements of vectors can be arranged in rows (row vectors) or columns (column vectors). The transpose of a row vector is that same sequence of elements presented in column format. Conversely, the transpose of a column vector is that same sequence of elements presented in row format. For example, $\{a_1, a_2, a_3\}^T = \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix}$.

A unit vector is a vector of length 1. A unit vector can be obtained from a vector v using the formula: $\hat{v} = \frac{v}{\|v\|_2}$. Unit vectors that are orthogonal are said to be orthonormal.

We can think of matrices as being an ordered set of column or row vectors. As a result, the terminology of matrices is closely related to that of vectors. The *column space* of a matrix is the vector space spanned by its columns. The *row space* of a matrix is the vector space spanned by its rows. The *rank* of a matrix is the number of linearly independent columns or rows. If $Rank[A] = n$, then the matrix has *full* column *rank*. If $Rank[A] < m, n$, then the matrix is *rank deficient*. An *n*-by-*n* matrix A is *singular* if the equation $Ax = 0$ has a solution when x is a non zero n dimensional vector. (We will discuss in detail the meaning of Ax later.) Finally, the *null space* of A is the set of all vectors x satisfying the equation $Ax = 0$.

Consider the following matrix:

$$A \in \mathbb{R}^{mxn} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$. We consider this matrix to have the basis

$e_m = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$. This matrix is designated the *m-dimensional identity matrix*. It is fundamental that $A = e_m A = A e_m$. In fact, we can consider this our most basic matrix factorization.

Just as we can change the coordinate system (also termed a basis) for describing a vector, we can re-express a matrix in an alternate basis. This process is a fundamental concept in matrix factoring.

We saw where the Euclidean norm for vectors gave us a measure of magnitude. In a similar way, the Frobenius norm of a matrix is defined as $\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij})^2 \right)^{1/2}$. We will further discuss matrix norms later in the book.

Matrices are often described by their shape, symmetry, and elemental population. For example, a matrix with many rows and few columns is said to be *tall*. A matrix with mostly zero elements is *sparse*. A matrix with all zeros below the principal diagonal is *upper triangular*.

Representing Matrices in C#.NET Arrays

Conventionally, we assign the first row or column of a matrix the index one (i.e.,. the element in the first row and column is a_{11}). Arrays in C# and many other programming languages are zero based. That is, the first element in the array is MyArray(0, 0). This nuisance is easily resolved by ignoring the first row and column of arrays. When we populate an array with a matrix, we simply begin by entering a_{11} in MyArray(1, 1) and we proceed from there. We declare a C# array for an m-by-n matrix with double[,] MyArray = new double[m + 1, n + 1].

When we declare an array — double[,] MyArray — we bring it into existence. When we dimension an array — double[,] MyArray = new double[m + 1, n + 1] — we set its size. In reality we are establishing a pointer for the array's contents to an appropriately sized section of memory. If we are going to set an array equal to another array, then we can avoid having to evaluate the proper dimensions and simply use the declaration approach. However, the statement ArrayA = ArrayB in fact simply sets the pointer for the arrays' content to the same place in memory, and any subsequent change in one will occur in the other. If you want two independent arrays, you will have to dimension the new array then copy using the statement Array.Copy(B, A, B.length). Be certain you understand the difference and carefully consider which approach to use. Both are indispensable for specific applications.

We can represent vectors as one dimensional arrays, and declare and size them with double[] Vector and double[] Vector = new double[m]. However, it is often unclear whether a vector is a column vector or a row vector when we use this approach.

Therefore, in this book — with a couple of exceptions — vectors are treated as special case 1-by-n (row vectors) or m-by-1 (column vectors) matrices

A method is a procedure, either a function or a subroutine, that will perform a specific task. A function explicitly returns a result. A subroutine can (but need not) return results by reference. When we develop methods for matrix computations we need to return values (usually arrays). So, we can return multiple arrays from a subprocedure by passing those arrays *by reference* as subprocedure arguments.

Another solution involves returning those arrays from a function as a *jagged array*. A jagged array is an array of arrays. Consider the following function:

```
  public double[][,] DO_IT( double[,] A )
        {
            double[][,] Result_Array = new double[2][,]; //Jagged array
            for function
            Result_Array[0] = ((System.Double[])(new double[7, 4 ]));
            Result_Array[1] = ((System.Double[])(new double[5, 3]));
            // Do some things that create two arrays to be returned
            then pack them into Result_Array(0) and Result_Array(1)
            return Result_Array;
         }
```

This function packs two arrays into one jagged array and returns the jagged array. On the consumer side we access the results with something like this:

```
private void Button1_Click( System.Object sender, System.EventArgs e )
        {
            A = new double[2, 2];
            A[1, 1] = 3;
            double[][,] Receiver = new double[2][,];
            Receiver = ((System.Double[][,])(DO_IT(A)));
            B = ((System.Double[,])(Receiver[0 ]));
            C = ((System.Double[,])(Receiver[1 ]));
         }
```

Be aware that there are some common language specification (CLS) issues with jagged arrays, and that classes using them should not be exposed to CLS-compliant code.

In this book we will pass multiple arrays using the subprocedure approach. Jagged arrays can be — in certain cases — more efficient, but code written with them is often hopelessly difficult to follow, and in this book following the code is the objective. Using the subprocedure with passing arrays by reference also allows the dot NET graphical user interface to provide inline help with subprocedure arguments. Here is a simple example of such a procedure together with an interface:

```
    public void DO_IT2( double[,] X, ref double[,] B, ref double[,] C)
        {
            // Do some things that create results in B and C
        }


    private void B1_Click( System.Object sender, System.EventArgs e )
        {
```

```
            A = new double[ 2, 2 ];
            A[ 1, 1 ] = 3;
            DO_IT2( A, ref B, ref C );
        }
```

Some additional points should be made before we move on. First, if you are using VB.NET, turn strict on. "[This] restricts implicit data type conversions to only widening conversions. [It] explicitly disallows any data type conversions in which data loss would occur and any conversion between numeric types and strings" (Microsoft VB Language Reference). Also, note in the code examples above that we are passing two dimensional arrays As Double(,) not As Array or As Object, thus giving our user (perhaps ourselves) a break from the problems of late binding.

The class library we will develop for matrix computations will have certain global variables. It is best to list these right now before we start adding methods to the class. Put a paper clip on this page or, better yet, enter these variable declarations in your programming environment now. Don't worry if you don't know their meaning or use.

```
    public class Matrix_Computation_Utility
      {

        private int[] Gauss_Encoded_P = new int[2]; // Row permutation
vector
        private int[] Gauss_Encoded_Q = new int[2]; // Column
permutation vector
        private int[] QR_Encoded_P = new int[2]; // QR column
permutation vector
        private int QRRank; // QR rank
        private int SVDRANK; // SVD Rank
        public double Machine_Error = 0.00000000000000011313; // unit
roundoff variable
        public double SVD_Rank_Determination_Threshold = 0.000001;//an
adjustable rank determination threshold
        public double Givens_Zero_Value = 0; // Threshold for Givens
process
        private int c_sign; // sign of the determinant
        public double HOUSE_BETA; // pipeline from house
        public double[] beta = new double[2]; // Vector of Householder
Beta values
        public delegate void doneEventHandler();
        public event doneEventHandler done; // Signal that a procedure
is complete
        // The following properties allow reading private globals.
        public int[] GE_Encoded_Row_Permutation
        {
            get
            {
                return Gauss_Encoded_P;
            }
        }
        public int[] GE_Encoded_Column_Permutation
        {
            get
            {
```

```
            return Gauss_Encoded_Q;
        }
    }
    public int[] QR_Encoded_Column_Permutation
    {
        get
        {
            return QR_Encoded_P;
        }
    }
    public int QR_Rank
    {
        get
        {
            return QRRank;
        }
    }
    public int svd_Rank
    {
        get
        {
            return SVDRANK;
        }
    }

}
```

Mathematical Operations

There are certain fundamental operations we must consider before we move on to our primary subject matter.

Transposition $C=A^T$

Transposing a matrix involves making the first column of the matrix the first row of the transpose, then making the second column the second row, and so on. Alternatively, we can understand the process as rotating the array 180º around the axis formed by the diagonal set of elements beginning with a_{11}. Note that the transposed matrix has n rows and m columns.

It is not always necessary to explicitly form a matrix's transpose. For example, if we want to perform an operation over i rows and j columns of A^T, then we can exchange the indices of our original matrix and perform that operation over j rows and i columns of A. See Matrix–Matrix Multiplication below for an example.

Implementation:

Given: $A \in \mathbb{R}^{mxn}$ determine $C \in \mathbb{R}^{nxm}$ such that $A^T = C$

```
public double[,] Transpose(double[,] Source_Matrix)
```

```
        {
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            int i = 0, j = 0;
            double[,] result = new double[n + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    result[j, i] = Source_Matrix[i, j];
                }
            }
            if (null != done) done();
            return result;
              }
```

Addition: $C = A + B$

When we add two matrices, we sum corresponding elements in each matrix. For example, suppose we are adding A and B to form C. The ij[th] element of C (C_{ij}) is $A_{ij} + B_{ij}$. Note that the matrices to be added must have the same dimensions.

Implementation:

Given: $A \in \mathbb{R}^{mxn}$, $B \in \mathbb{R}^{mxn}$ determine $C \in \mathbb{R}^{mxn}$ such that $C = A + B$

```
        public double[,] Matrix_Add(double[,] A_Matrix, double[,]
B_Matrix)
        {
            int m = A_Matrix.GetLength(0) - 1;
            int n = A_Matrix.GetLength(1) - 1;
            double[,] result = new double[m + 1, n + 1];
            if (m != (B_Matrix.GetLength(0) - 1) | n !=
(B_Matrix.GetLength(1) - 1))
            {
                throw (new ApplicationException("A + B requires the
number of rows and columns in A and B be equal"));
            }
            int i = 0, j = 0;
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    result[i, j] = A_Matrix[i, j] + B_Matrix[i, j];
                }
            }
            if (null != done) done();
            return result;
              }
```

Scalar Multiplication

Here we simply multiply every element by a scalar to form a new matrix with the same size and shape as the original matrix.

Implementation:

Given: $A \in \mathbb{R}^{mxn}$ determine $C \in \mathbb{R}^{mxn}$ such that $C = \alpha A$ where α is a scalar.

```
        public double[,] Scalar_multiply(double[,] A_Matrix, double
Scalar)
        {
            int m = A_Matrix.GetLength(0) - 1;
            int n = A_Matrix.GetLength(1) - 1;
            double[,] result = new double[m + 1, n + 1];
            int i = 0, j = 0;
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    result[i, j] = Scalar * A_Matrix[i, j];
                }
            }
            if (null != done) done();
            return result;
        }
```

Matrix–Matrix Multiplication

With transposition running a close second, matrix multiplication is the most common elementary operation we will encounter. It is a costly process and it is fundamental that we implement multiplication procedures strategically. The following schematic illustrates the multiplication process.

$$\begin{bmatrix} a_{11} & \cdots & a_{1p} \\ \vdots & \ddots & \vdots \\ a_{i1} & \cdots & a_{ip} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mp} \end{bmatrix} \begin{bmatrix} b_{11} & \cdots & b_{1j} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{pj} & \cdots & b_{pn} \end{bmatrix} =$$

Column J

$$\text{Row i} \begin{bmatrix} a_{11}b_{11}+\ldots+a_{1p}b_{p1} & \cdots & \vdots & \cdots & a_{11}b_{1n}+\ldots+a_{1p}b_{pn} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \vdots & \ddots & a_{i1}b_{1j}+\ldots+a_{ip}b_{pj} & \ddots & \vdots \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1}b_{11}+\ldots+a_{mp}b_{p1} & \cdots & \vdots & \cdots & a_{m1}b_{1n}+\ldots+a_{mp}b_{pn} \end{bmatrix}$$

As you can see, the ijth element of C is the dot product of the ith row of A and the jth column of B. Clearly, the number of columns in A must be equal to the number of rows in B.

Implementation:

Given: $A \in \mathbb{R}^{mp}, B \in \mathbb{R}^{pn}$ determine $C \in \mathbb{R}^{mn}$ such that $C = AB$

```
        public double[,] Matrix_Multiply(double[,] A_Matrix, double[,]
B_Matrix)
        {
            int m = A_Matrix.GetLength(0) - 1;
            int n = B_Matrix.GetLength(1) - 1;
            int p = A_Matrix.GetLength(1) - 1;
            double[,] result = new double[m + 1, n + 1];
            if (p != (B_Matrix.GetLength(0) - 1))
            {
                throw (new ApplicationException("for A X B, columns in
A and rows in B must be equal"));
            }
            int i = 0, j = 0, k = 0;
            for (i = 1; i <= m; i++)
            {
                for (k = 1; k <= p; k++)
                {
                    for (j = 1; j <= n; j++)
                    {
                        result[i, j] = A_Matrix[i, k] * B_Matrix[k, j]
+ result[i, j];
                    }
                }
            }
            if (null != done) done();
            return result;
        }
```

Matrix–matrix multiplication is not commutative**,** i.e., it is generally the case that AB ≠ BA. It is, however, always the case — and this is important — that $AB = \left(B^T A^T\right)^T$.

If we represent two column vectors v and u as matrices, then the dot product is $< v, u >= v^T u = u^T v$. If we represent two row vectors v and u as matrices, then the dot product is $< v, u >= vu^T = uv^T$.

The order of the loops in the multiplication algorithm can be changed from the above strategy to: For i = 1 To m: For j=1 To n: For k = 1 To p. The form of the calculation stays the same, namely C(i, j) = A (i, k) * B (k, j) + C(i, j). In fact, there are six loop sequences that will work: i,j,k; j,i,k; i,k,j; j,k,i; k,i,j; and k,j,i. . However, the data access properties will change and some variants will be more efficient than others. Consider the i,k,j variant used in the implementation. The inner loop accesses the elements of Result(i, j) and B_Matrix(k, j) sequentially along the current row (i for Result and k for B_Matrix). The inner loop data access for the six variants is detailed below.

Variant	Inner Loop Data Access
i,j,k	A rows, B columns
j,i,k	A rows, B columns
i,k,j	B rows, C rows
j,k,i	A columns, C columns
k,i,j	B rows, C rows
k,j,i	A columns, C columns

On all of the computers I have tested (both Athalon and Intel processor based, all running a VB.NET test program on Windows 9X or XP), I have found that the i,k,j and k,i,j variants run nearly twice as fast as the j,k,i and k,j,i variants (the i,j,k and j,i,k are in-between). Relative standard deviations are all less than 1%. The effect is most noticeable and meaningful when evaluated using a fully deployed release test application. These results indicate that arrays are being stored sequentially element-by-element along each row on the systems I am evaluating. Using our indexing convention of Array(row, col), the behavior is referred to as *row major* (as opposed to *column major*) array storage. Because the time difference is significant, your implementation should be tailored to your target machine.

Say we want to obtain the result $C = AB^T$ where $A \in \square^{mxp}$ and $B \in \square^{nxp}$. One way to approach the problem is to explicitly form B^T then multiply it by A. Another more efficient way is to implement the following algorithm:

```
for i = 1 to m
   for j = 1 to p
      for k = 1 to n
         C(i,k)=A(i,j) * B(k,j) + C(i,k)
next: next: next
```

This approach, which avoids the explicit formation of B^T, will return the same result. We will call this form of shortcut an *implicit implementation*, and we will use this frequently not only to reduce computational overhead but, as we will see later, to improve accuracy.

With the increasing use of hyper-threading and dual core processors, it may make sense to split large multiplications into independent threads. For example, using the i,k,j variant, we can split A(i, k) into A(q, k) and A(r, k), where q ranges from 1 to floor(i/2) and r ranges from floor(i/2+1) to m. We can solve C(q, j) and C(r, j) independently, then recombine them for C(i, j):

```
        public double[,] Parallel_multiply(double[,] A_Matrix,
double[,] B_Matrix)
        {
            int i = 0, j = 0;
            int m = A_Matrix.GetLength(0) - 1;
            int n = B_Matrix.GetLength(1) - 1;
            int p = A_Matrix.GetLength(1) - 1;
            if (p != (B_Matrix.GetLength(0) - 1))
```

```
            {
                throw (new ApplicationException("for A X B, columns in
A and rows in B must be equal"));
            }
            double[,] c = new double[m + 1, n + 1];
            int q =
System.Convert.ToInt32(System.Math.Floor(System.Convert.ToSingle(m) /
2));
            double[,] A1 = new double[q + 1, p + 1];
            double[,] A2 = new double[m - q + 1, p + 1];
            for (i = 1; i <= q; i++)
            {
                for (j = 1; j <= p; j++)
                {
                    A1[i, j] = A_Matrix[i, j];
                }
            }
            for (i = q + 1; i <= m; i++)
            {
                for (j = 1; j <= p; j++)
                {
                    A2[i - q, j] = A_Matrix[i, j];
                }
            }
            CoMultiply t1 = new CoMultiply();
            t1.a1 = A1;
            t1.a2 = A2;
            t1.b = B_Matrix;
            System.Threading.Thread thread1 = new
System.Threading.Thread(new
System.Threading.ThreadStart(t1.multiply1));
            System.Threading.Thread thread2 = new
System.Threading.Thread(new
System.Threading.ThreadStart(t1.multiply2));
            thread1.Start();
            thread2.Start();
            thread1.Join();
            thread2.Join();
            double[,] c1 = t1.c1;
            double[,] c2 = t1.c2;
            c = new double[m + 1, n + 1];
            for (i = 1; i <= q; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    c[i, j] = c1[i, j];
                }
            }
            for (i = q + 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    c[i, j] = c2[i - q, j];
                }
            }
            if (null != done) done();
            return c;
```

```
        }

    public class CoMultiply
    {
        public double[,] a1;
        public double[,] a2;
        public double[,] b;
        public double[,] c1;
        public double[,] c2;
        public void multiply1()
        {
            int m = a1.GetLength(0) - 1;
            int n = b.GetLength(1) - 1;
            int p = a1.GetLength(1) - 1;
            c1 = new double[m + 1, n + 1];
            if (p != (b.GetLength(0) - 1))
            {
                throw (new ApplicationException("for A X B, columns in
A and rows in B must be equal"));
            }
            int i = 0, j = 0, k = 0;
            for (i = 1; i <= m; i++)
            {
                for (k = 1; k <= p; k++)
                {
                    for (j = 1; j <= n; j++)
                    {
                        c1[i, j] = a1[i, k] * b[k, j] + c1[i, j];
                    }
                }
            }
        }

        public void multiply2()
        {
            int m = a2.GetLength(0) - 1;
            int n = b.GetLength(1) - 1;
            int p = a2.GetLength(1) - 1;
            c2 = new double[m + 1, n + 1];
            if (p != (b.GetLength(0) - 1))
            {
                throw (new ApplicationException("for A X B, columns in
A and rows in B must be equal"));
            }
            int i = 0, j = 0, k = 0;
            for (i = 1; i <= m; i++)
            {
                for (k = 1; k <= p; k++)
                {
                    for (j = 1; j <= n; j++)
                    {
                        c2[i, j] = a2[i, k] * b[k, j] + c2[i, j];
                    }
                }
            }
        }
```

```
}
```

Be aware that a mutual data bus is being shared on these processors and processor capacity utilization may be strikingly hindered by inadequate data transfer capacity.

Floating Point Numbers and Operations Counts

Real numbers, as we discussed, can be represented as points on a number line. There are an infinite number of these points in any segment of the line. A computer does not have an infinitely high resolution, so it can only work with a (finite) subset of the real numbers. Operations on these numbers are termed *floating point operations*. As an artifact of limited resolution, computers have a machine dependent error (or round off error) that we will denote with ε.

A phenomenon known as catastrophic cancellation occasionally occurs when large values are added or subtracted to produce small values with significantly diminished precision. Watkins provides a very intuitive and detailed discussion of this effect.

If we denote any of the four basic arithmetic floating point operations with $\circ$ and the corresponding real operations with $\bullet$, then: $a \circ b = (a \bullet b)(1 + \varphi)$ for any $|\varphi| \leq \varepsilon$. In other words, a rounding error occurs. One can determine a functional value for machine precision that is tolerant of differing implementations of floating point standards with the following procedure:

```
        public void Find_precision()
        {
            // At what threshold value of delta does 1 + delta no
longer equal 1?
            double delta = 0;
            for (delta = 1.0E-18; delta <= 1; delta += 1.0E-18)
            {
                if (1 + delta != 1)
                {
                    break;
                }
            }
            Machine_Error = 1.01 * delta; // Global, small multiple of
unit roundoff
        }
```

Consider matrix multiplication:

For i = 1 To m: For k = 1 To p: For j = 1 To n
    C(i, j) = A (i, k) * B (k, j) + C(i, j)
Next: Next: Next

The central calculation includes one floating point multiplication and one floating point addition, i.e., two floating point operations. This calculation is performed mpn times. The total number of floating point operations for matrix–matrix multiplication is 2mpn. Some sources, including this book, abbreviate floating point operations with the acronym FLOPs. Be aware that other sources reserve this acronym for floating point operations *per second* (a measure of speed), and still others refer to a floating point add/multiply pair as a FLOP. We will not discuss floating point operations per second other than to say the concept is essentially meaningless without a reference standard algorithm for comparing machine-to-machine performance. In a straight loop with two numbers being added, such as

```
For i = 1 to 1000: For j = 1 to 1000: For k = 1 to 1000
X = 5.02 + 3.01
Next: Next: Next
```

most PCs will score around 0.5 to 2 billion floating point operations per second. With a matrix–matrix multiplication where data in arrays must be accessed, the speed is highly dependent on algorithm implementation and is typically 50 to 200 million floating point operations per second.

Linear Systems in Matrix Form

Consider the following system of linear equations:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + a_{12}x_3 &= b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{22}x_3 &= b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{32}x_3 &= b_3
\end{aligned}
$$

We can place the coefficients a_{ij} in a 3 × 3 matrix and factor out the x vector as a column vector $\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$. You should convince yourself of this form by reviewing the definition of matrix multiplication and considering a column vector to be a matrix with one column. We use the following notation, $Ax = b$, to denote the factorization given above. If A is square, then the system is said to be determined. If for $A \in \mathbb{R}^{mn}$ $m > n$, then the system is said to be over determined. If $m < n$, then the system is under determined. Over determined systems have an infinity of solutions, of which at least one is a minimum norm solution. Under determined systems have either no solutions or an infinity of solutions — again, at least one of which is a minimum norm solution.

2. Gaussian Elimination, the LU Factorization, and Determined Systems

Gaussian Elimination

Upper Triangulation Algorithms

We frequently encounter Gaussian elimination in the analysis of linear systems. With Gaussian elimination we convert a matrix of form:

$$\begin{pmatrix} a_{11} & \cdots & a_{1m} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} & \cdots & a_{mn} \end{pmatrix}, \text{m}\leq\text{n}$$

to one of form:

$$\begin{pmatrix} \hat{a}_{11} & \cdots & \cdots & \hat{a}_{1m} & \cdots & \hat{a}_{1n} \\ 0 & \ddots & \ddots & \vdots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \hat{a}_{mm} & \cdots & \hat{a}_{mn} \end{pmatrix}$$

We are zeroing the elements below the principal diagonal of the matrix's leftmost square. The matrix may be square, in which case the leftmost square is the matrix itself. For the purpose of our discussion, the matrix cannot have more rows than columns.

The procedure begins with obtaining a vector by multiplying the first row by the factor $\frac{a_{2,1}}{a_{1,1}}$ (row factor 2, 1), then subtracting that vector from the second row. This places a zero in the $a_{2,1}$ position. Next, we obtain a vector by multiplying the first row by the factor $\frac{a_{3,1}}{a_{1,1}}$ (row factor 3, 1), then we subtract that vector from the third row. This procedure is repeated for the remaining rows, then column two is processed in the same way beginning with row factor 3, 2. Our denominator for the row factor is always the diagonal element of the column we are working on, and the numerator is the element we seek to zero. Frequently the resultant matrix is computed in the form of an encoded matrix, where the zeros below the principal diagonal are replaced with the row factors used to produce a zero in that position.

Note that the left square of the matrix is being upper triangulated. The following code encapsulates the process.

```
public double[,] GaussUT_nopiv(double[,] Source_Matrix)
```

```
        {
            // no pivot upper triangular Gauss elimination of leftmost
square of matrix
            // overwrites Source_Matrix, returning encoded array with
rowfactors in lower triangle
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            int i = 0, j = 0, k = 0;
            double rowfactor = 0;
            if (m > n)
            {
                throw new ApplicationException("Rows exceeds columns.
Leftmost square undefined");
            }
            // gaussian elimination loop
            for (k = 1; k <= m; k++)
            {
                if (Source_Matrix[k, k] == 0)
                {
                    throw new ApplicationException("Divide by zero
occurs, consider pivoting.");
                }
                // the loop to zero the lower triangle (actually
rowfactors are stored in the lower triangle)
                for (i = k + 1; i <= m; i++)
                {
                    rowfactor = Source_Matrix[i, k] / Source_Matrix[k,
k];
                    for (j = k; j <= n; j++)
                    {
                        Source_Matrix[i, j] = Source_Matrix[i, j] -
rowfactor * Source_Matrix[k, j];
                    }
                    Source_Matrix[i, k] = rowfactor;
                }
            }
            if (null != done) done();
            return Source_Matrix;
        }
```

Obviously a problem arises when any given row factor's denominator is zero. Also, arithmetic reliability problems occur when the row factor's denominator is disproportionately low in absolute value compared to other matrix elements (rounding errors occur). The solution lies in a technique called pivoting.

There are algorithms for partial pivoting and full pivoting. Let us first consider row pivoting. Initially we have the matrices

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1m} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} & \cdots & a_{mn} \end{pmatrix} \text{ and } Prow = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{pmatrix}, \text{ where } Prow \text{ is an}$$

identity matrix with the same dimensions as the leftmost square of A (m × m).

Again, we seek to triangulate the leftmost square of A, and we work from the diagonal elements down for any given column. The row containing the diagonal element of interest is called the pivot row. We compare the absolute value of the diagonal element with that of each of the elements below it. The row with the element of highest absolute value is swapped with the pivot row (if indeed the pivot row is not already the row with the highest absolute value element). We keep track of row exchanges throughout the computation with an encoded row permutation matrix. (Gauss_Encoded_P. Permutation matrices and encoded permutation matrices are discussed in detail later in this chapter.) Here is a procedure for doing this:

```
        public double[,] GaussUT_partial_piv(double[,] Source_Matrix)
        {
            // Row pivot upper triangular Gauss elimination of leftmost
square of matrix
            // returns encoded array with rowfactors in lower triangle.
Modifies global Prow, overwrites source_matrix
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            if (m > n)
            {
                throw new ApplicationException("Rows exceeds columns.
Leftmost square undefined");
            }
            Gauss_Encoded_P = new int[m + 1];
            c_sign = 1; // This is here for determinant calculations.
            int i = 0, j = 0, k = 0, exrow = 0, temp2 = 0;
            double temp1 = 0, rowfactor = 0, amax = 0;
            // initialize permutation matrices
            for (i = 1; i <= m; i++)
            {
                Gauss_Encoded_P[i] = i;
            }
            // gaussian elimination loop
            for (k = 1; k <= m; k++)
            {
                exrow = k;
                amax = System.Math.Abs(Source_Matrix[k, k]); // find
the element with the highest abs value below the pivot and record its
row
                for (i = k; i <= m; i++)
                {
                    if (System.Math.Abs(Source_Matrix[i, k]) > amax)
                    {
                        amax = System.Math.Abs(Source_Matrix[i, k]);
                        exrow = i;
                    }
                }
                if (exrow != k)
                { // exchange rows

                    for (j = 1; j <= n; j++)
                    { // for the source_matrix matrix

                        temp1 = Source_Matrix[k, j];
                        Source_Matrix[k, j] = Source_Matrix[exrow, j];
```

```
                        Source_Matrix[exrow, j] = temp1;
                    }
                    temp2 = Gauss_Encoded_P[k]; // for the permutation
matrix
                    Gauss_Encoded_P[k] = Gauss_Encoded_P[exrow];
                    Gauss_Encoded_P[exrow] = temp2;
                    c_sign = c_sign * -1;
                }
                if (Source_Matrix[k, k] == 0)
                {
                    throw new ApplicationException("Singular matrix");
                }
                // the loop to zero the lower triangle (actually
rowfactors are stored in the lower triangle)
                for (i = k + 1; i <= m; i++)
                {
                    rowfactor = Source_Matrix[i, k] / Source_Matrix[k,
k];
                    for (j = k; j <= n; j++)
                    {
                        Source_Matrix[i, j] = Source_Matrix[i, j] -
rowfactor * Source_Matrix[k, j];
                    }
                    Source_Matrix[i, k] = rowfactor;
                }
            }
            if (null != done) done();
            return Source_Matrix;
        }
```

The full pivoting procedure considers the diagonal element of focus at any step of the process to be a pivot point. At any step K we begin at the diagonal element and search and compare each element of the A pivot submatrix to determine which has the highest absolute value. Then row and column exchanges are performed to bring the element of highest absolute value to the pivot point. We keep track of the row and column exchanges throughout the computation with encoded row and column permutation matrices Gauss_Encoded_P and Gauss_Encoded_Q.

The schematic below shows what is meant by the pivot submatrix. At any point in the procedure only elements within this submatrix are compared, but the exchanges involve the entire row or column.

$$\begin{bmatrix} \hat{a}_{11} & \cdots & \cdots & \cdots & \hat{a}_{1n} \\ 0 & \ddots & & \ddots & \vdots \\ \vdots & 0 & \hat{a}_{kk} & \cdots & \hat{a}_{kn} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \hat{a}_{mk} & \cdots & \hat{a}_{mn} \end{bmatrix}$$

Initially we have the matrices $A = \begin{pmatrix} a_{11} & \cdots & a_{1m} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} & \cdots & a_{mn} \end{pmatrix}$ and $Prow = Pcol = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 \end{pmatrix}$, where $Prow$ and $Pcol$ are identity matrices with the same dimensions as the leftmost square of A. Here is the procedure:

```
        public double[,] GaussUT_full_piv(double[,] Source_Matrix)
        {
            // Full pivot upper triangular Gauss elimination of
leftmost square of matrix
            // returns encoded array with rowfactors in lower triangle.
Modifies global Gauss_Encoded_Q and Gauss_Encoded_P, overwrites
source_matrix
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            if (m > n)
            {
                throw new ApplicationException("Rows exceeds columns.
Leftmost square undefined");
            }
            Gauss_Encoded_P = new int[m + 1];
            Gauss_Encoded_Q = new int[n + 1];
            c_sign = 1;
            int i = 0, j = 0, k = 0, exrow = 0, excol = 0, temp2 = 0;
            double temp1 = 0, rowfactor = 0, amax = 0;
            // initialize permutation matrices
            for (i = 1; i <= m; i++)
            {
                Gauss_Encoded_P[i] = i;
                Gauss_Encoded_Q[i] = i;
            }
            // gaussian elimination loop
            for (k = 1; k <= m; k++)
            {
                exrow = k;
                excol = k;
                amax = System.Math.Abs(Source_Matrix[k, k]); // find
the element with the highest abs value in the pivot submatrix and
record its row and column
                for (i = k; i <= m; i++)
                {
                    for (j = k; j <= m; j++)
                    {
                        if (System.Math.Abs(Source_Matrix[i, j]) >
amax)
                        {
                            amax = System.Math.Abs(Source_Matrix[i,
j]);
                            exrow = i;
```

```
                        excol = j;
                    }
                }
            }
            if (exrow != k)
            { // exchange rows
                for (j = 1; j <= n; j++)
                { // for the source_matrix matrix
                    temp1 = Source_Matrix[k, j];
                    Source_Matrix[k, j] = Source_Matrix[exrow, j];
                    Source_Matrix[exrow, j] = temp1;
                }
                temp2 = Gauss_Encoded_P[k]; // for the permutation
matrix
                Gauss_Encoded_P[k] = Gauss_Encoded_P[exrow];
                Gauss_Encoded_P[exrow] = temp2;
                c_sign = c_sign * -1;
            }
            if (excol != k)
            { // exchange columns (to bring element with highest
abs value to the principal diagonal
                for (i = 1; i <= m; i++)
                { // for the source_matrix matrix
                    temp1 = Source_Matrix[i, k];
                    Source_Matrix[i, k] = Source_Matrix[i, excol];
                    Source_Matrix[i, excol] = temp1;
                }
                temp2 = Gauss_Encoded_Q[k]; // for the permutation
matrix
                Gauss_Encoded_Q[k] = Gauss_Encoded_Q[excol];
                Gauss_Encoded_Q[excol] = temp2;
                c_sign = c_sign * -1;
            }
            if (Source_Matrix[k, k] == 0)
            {
                throw new ApplicationException("Singular matrix");
            }
            // the loop to zero the lower triangle (actually
rowfactors are stored in the lower triangle)
            for (i = k + 1; i <= m; i++)
            {
                rowfactor = Source_Matrix[i, k] / Source_Matrix[k,
k];
                for (j = k; j <= n; j++)
                {
                    Source_Matrix[i, j] = Source_Matrix[i, j] -
rowfactor * Source_Matrix[k, j];
                }
                Source_Matrix[i, k] = rowfactor;
            }
        }
        if (null != done) done();
        return Source_Matrix;
    }
```

Lower Triangulation Algorithms

There is nothing sacred about making the leftmost square upper triangular as opposed to lower triangular. Consider the following function for lower triangulation without pivoting:

```
        public double[,] GaussLT_nopiv(double[,] Source_Matrix)
        {
            // no pivot lower triangular Gauss elimination of leftmost
square of matrix
            // no rowfactors are storred, overwrites original matrix
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            int i = 0, j = 0, k = 0;
            double rowfactor = 0;
            if (m > n)
            {
                throw new ApplicationException("Rows exceeds columns.
Leftmost square undefined");
            }
            for (k = m; k >= 2; k += -1)
            {
                for (i = k - 1; i >= 1; i += -1)
                {
                    rowfactor = Source_Matrix[i, k] / Source_Matrix[k,
k];
                    for (j = n; j >= 1; j += -1)
                    {
                        Source_Matrix[i, j] = Source_Matrix[i, j] -
rowfactor * Source_Matrix[k, j];
                    }
                }
            }
            if (null != done) done();
            return Source_Matrix;
        }
```

In this procedure the row factors are not being stored (you will see why later), but they certainly could be. We can add steps to perform row pivoting or full pivoting, but there is a catch. Note that in the upper triangulation algorithms, the step to zero the lower triangle has the form:

```
For i = k + 1 To m
   rowfactor = A(i, k) / A(k, k)
   For j = k To n
      A(i, j) = A(i, j) - rowfactor * A(k, j)
   Next j
   A(i, k) = rowfactor
Next i
```

The inner loop (For j = k To n; … Next j) only needs to execute from k to n because A(k, p) is zero for all p < k. With lower triangulation for $m < n$, the zeros occur

bounded by non-zero values to their left and right. We either must perform the inner loop for the entire row (as above), or employ some "skip-over" strategy — each of which results in an unnecessary loss of efficiency. So, except when lower triangulation is required by some special application, upper triangulation seems to work best.

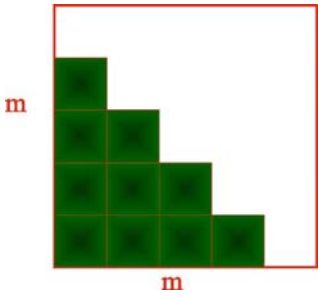
Choosing a Pivot Strategy

We have investigated implementations of Gaussian elimination algorithms, but we have not yet used them to obtain solutions. It makes sense at this point to discuss computational overhead for the various Gaussian elimination procedures, and to make some generalizations about how the procedures compare in terms of accuracy. An accurate answer is never guaranteed with Gaussian elimination (nor with any other matrix computation). The worst case scenario is the case of the singular matrix, but at least in this case the algorithm throws an exception (unless rounding errors represent zero values with very low but non-zero numbers). Nearly singular matrices, or "ill conditioned" matrices, also produce inaccurate results. This type of error and how to predict it is discussed later. The accuracy we consider here is that which is attributable to the way elements are arranged in the matrix and not to the inherent condition of the matrix. Loss of accuracy due to inherent algorithmic shortcomings is generally referred to as algorithmic instability. Often — and specifically here with Gaussian elimination — evaluating which algorithm to use reduces to evaluating the acceptable speed vs. accuracy trade-off.

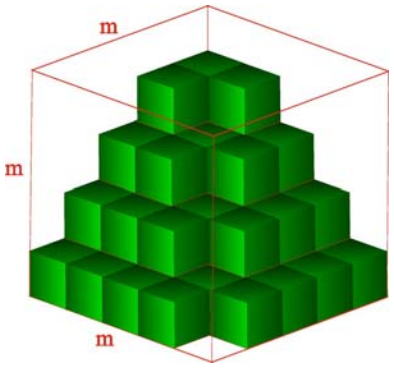
Gaussian elimination without pivoting is generally a bad idea. Consider its use only if you are certain (in all applications using your code) that zeros or unacceptably low values will never haphazardly find their way onto the principal diagonal of your matrix's leftmost square. Always err on the side of caution. (See Golub/Van Loan 3.4.10 for a discussion of avoiding pivoting.)

All three forms presented for Gaussian elimination require the same number of floating point operations ($2m^3/3$ for a square matrix), but pivoting requires additional computational overhead. When row pivoting is performed, comparison steps are added to the algorithm. Initially, the absolute value of the element on the principal diagonal of a given column is stored in a variable "amax". This value is then compared with each element's absolute value below it, and amax is replaced by any value exceeding the existing value. If there are m rows in a matrix there will be $m-1$ comparisons for the first column. There is no need to re-evaluate which element has the highest absolute value once this initial determination has been made for a given column. The second column will have $m-2$ comparisons. The overall number of comparisons is: $\sum_{i=1}^{m-1}(m-i)$.

The diagram below schematically shows the elements being compared. They occupy the area under the diagonal of the matrix. The number of comparisons is proportional to — and not greater than — m^2, the area of the square. Many textbooks state this measure of computational overhead with "big O" notation, i.e., the number of comparisons is $O(m^2)$.



As with row pivoting, in the full pivot Gaussian elimination algorithm the absolute value of the element on the principal diagonal of a given column is stored in a variable "amax". This value is now compared using the same strategy as the row pivot, except we use a search and compare procedure covering the entire submatrix. For the first column, there is a total of $m^2 - 1$ comparisons (it is minus one because we do not compare the first element with itself). The next column considers a pivot submatrix with dimensions $m-1$ by $m-1$, so the number of comparisons is $(m-1)^2 - 1$. The overall number of comparisons is: $\sum_{i=1}^{m-1}\left((m-i)^2 - 1\right)$. The schematic below is a pyramidal structure enveloped by a cube of volume m^3. The base of the pyramid contains the elements being compared for column one's pivot point. Similarly, the ascending layers correspond to the pivot submatrices for the remaining columns. The number of comparisons is proportional to — and not greater than — m^3, the volume of the cube. The number of comparisons is $O(m^3)$.



Increased pivoting requires other expensive procedures, such as maintaining permutation matrices and swapping rows or rows and columns. As a result, there is a substantial increase in computational overhead as we progress from a non-pivoting algorithm to a full pivoting algorithm. The gain in the algorithm's ability to give the right answer is substantial when we choose row pivoting over no pivoting. The gain in stability when we move to full pivoting from row pivoting is negligible for most practical purposes. However, if waiting for the result is not an issue, full pivoting is the most cautious approach.

Determinants

Although determinants are rarely used in linear solutions algorithms, we should briefly discuss them because they are used in the Cramer's rule method for solving determined systems. Determinants apply to square matrices. One of several methods for evaluating determinants involves Gaussian elimination with pivoting. As usual we arrive at a matrix of form $\begin{pmatrix} a_{11} & \cdots & \cdots & a_{1m} \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{mm} \end{pmatrix}$. The determinant is given by $Sign\prod_{k=1}^{m} a_{k,k}$ (the product of the diagonal elements). *Sign* is negative whenever the number of row exchanges plus column exchanges is odd.

For the encapsulation below, recall that the GaussUT_full_piv function utilizes c_sign, a variable with class scope:

```
        public double Determinant(double[,] Source_Matrix)
        {
            double determinantReturn = 0;
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            if (m != n)
            {
                throw new ApplicationException("matrix must be
square");
            }
            int i = 0;
            double[,] arrTemp = new double[m + 1, n + 1];
            Array.Copy(Source_Matrix, arrTemp, arrTemp.Length);
            Array.Copy(GaussUT_full_piv(arrTemp), arrTemp,
arrTemp.Length);
            determinantReturn = 1;
            for (i = 1; i <= m; i++)
            {
                determinantReturn = determinantReturn * arrTemp[i, i];
            }
            determinantReturn = determinantReturn * c_sign;
            if (null != done) done();
            return determinantReturn;
        }
```

Inversion by Augmentation

Another fundamental computation is matrix inversion. Inversion only applies to square matrices. Here we consider:

$A \in \mathbb{R}^{mxm}$ and $A^{-1} \in \mathbb{R}^{mxm}$

such that: $AA^{-1} = A^{-1}A = I$ where I is the m-dimensional identity matrix.

There are many algorithms for obtaining A^{-1}. The inversion by augmentation technique employs Gaussian elimination. Row pivoting or full pivoting Gaussian elimination can also be used. I am introducing this algorithm as a precursor to a later discussion, and I will not be providing an implementation. I will present a better algorithm and its implementation after we discuss the LU factorization.

First we form an augmented matrix. We take $A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$ and extend the matrix by adding columns to the right representing the appropriately dimensioned identity matrix:

$$\text{A augmented} = \begin{bmatrix} a_{11} & \cdots & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots & 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & a_{nn} & 0 & \cdots & \cdots & 1 \end{bmatrix}$$

Next a full pivot Gaussian elimination strategy is used to zero the lower triangle of the left square.

$$\text{A augmented} = \begin{bmatrix} \hat{a}_{11} & \cdots & \cdots & \hat{a}_{1n} & b_{11} & \cdots & \cdots & b_{1n} \\ 0 & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \hat{a}_{nn} & b_{n1} & \cdots & \cdots & b_{nn} \end{bmatrix}$$

Here b and $\hat{a}$ denote transitional values. The row factors are deleted from the result matrices. We will not need them.

Next we zero the leftmost square's upper triangle using Gaussian elimination, this time using the lower triangulation procedure (without pivoting).

$$\text{A augmented} = \begin{bmatrix} \hat{a}_{11} & 0 & \cdots & 0 & c_{11} & \cdots & \cdots & c_{1n} \\ 0 & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ddots & 0 & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \hat{a}_{nn} & c_{n1} & \cdots & \cdots & c_{nn} \end{bmatrix}$$

We divide each row by the corresponding diagonal element in the leftmost square and obtain:

$$\text{A augmented} = \left[\begin{array}{cccc|cccc} 1 & 0 & \cdots & 0 & a^{-1}_{11} & \cdots & \cdots & a^{-1}_{1n} \\ 0 & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ddots & 0 & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1 & a^{-1}_{n1} & \cdots & \cdots & a^{-1}_{nn} \end{array}\right]$$

The rightmost square is now a permutation of the inverse matrix, which we parse into its own n-by-n matrix. This matrix is multiplied by the Column Permutation matrix obtained from the full pivot Gaussian elimination step to obtain A^{-1}.

The LU Factorization

We have actually already studied the LU factorization. Its essence is that of upper triangulation by Gaussian elimination, and all that remains to be discussed is the simple formality of nomenclature. In our implementations of the upper triangulation Gaussian elimination algorithms, the row factors are stored in the lower left triangle below the principal diagonal of the leftmost square of the matrix. Also, encoded row and column permutation matrices are being calculated and stored as global arrays Gauss_Encoded_P and Gauss_Encoded_Q. With the LU factorization we restrict consideration to square matrices, so the leftmost square is the matrix itself.

First we execute any of the three upper triangulation Gaussian elimination functions on a matrix A — GaussUT_nopiv(A), GaussUT_partial_piv(A), or GaussUT_full_piv(A). Next, the row factors (elements below the diagonal) are parsed into a new matrix L which is unit diagonal and square

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ RF_{21} & 1 & 0 & 0 \\ \vdots & \ddots & 1 & 0 \\ RF_{m1} & \cdots & RF_{mn-1} & 1 \end{pmatrix}, \text{ where RF is the row factor.}$$

The upper triangular portion of the Gaussian elimination matrix is parsed into U.

$$U = \begin{bmatrix} \hat{a}_{11} & \cdots & \cdots & \hat{a}_{1m} \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \hat{a}_{mm} \end{bmatrix}$$

Finally, for the sake of clarity, let us call the row and column permutation matrices P and Q respectively. L, U, P and Q make up the LU decomposition and $PAQ = LU$. When row pivoting is used, Q is an m × m identity matrix.

The following two functions take as an argument a square matrix and perform both the Gaussian elimination and subsequent parsing. P and Q are stored as encoded matrices in the global variables Gauss_Encoded_P and Gauss_Encoded_Q, and they can be accessed through the read only properties GE_Encoded_Row_Permutation and GE_Encoded_Column_Permutation.

Row Pivot LU

```
        public void LU_row_pivot(double[,] Source_Matrix, ref double[,]
L, ref double[,] U)
        {
            // This is mostly a simple parsing algorithm. The actual LU
is performed by GaussUT_partial_piv.
            // Modifies global Gauss_Encoded_Q and Gauss_Encoded_P,
overwrites source_matrix with encoded LU
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            if (m != n)
            {
                throw new ApplicationException("matrix must be
square");
            }
            L = new double[m + 1, n + 1];
            U = new double[m + 1, n + 1];
            int i = 0, j = 0;
            Source_Matrix = GaussUT_partial_piv(Source_Matrix);
            // parse encoded_array to L and U
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    if (i > j)
                    {
                        L[i, j] = Source_Matrix[i, j];
                    }
                    if (i < j)
                    {
                        L[i, j] = 0;
                    }
                    if (i == j)
                    {
                        L[i, j] = 1;
                    }
                    if (i <= j)
                    {
                        U[i, j] = Source_Matrix[i, j];
                    }
                }
            }
            if (null != done) done();
        }
```

Full Pivot LU

```
        public void LU_full_pivot(double[,] Source_Matrix, ref
double[,] L, ref double[,] U)
        {
            // This is mostly a simple parsing algorithm. The actual LU
is performed by GaussUT_full_piv.
            // Modifies global Gauss_Encoded_Q and Gauss_Encoded_P,
overwrites source_matrix with encoded LU
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            if (m != n)
            {
                throw new ApplicationException("matrix must be
square");
            }
            L = new double[m + 1, n + 1];
            U = new double[m + 1, n + 1];
            int i = 0, j = 0;
            Source_Matrix = GaussUT_full_piv(Source_Matrix);
            // parse encoded_array to L and U
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    if (i > j)
                    {
                        L[i, j] = Source_Matrix[i, j];
                    }
                    if (i < j)
                    {
                        L[i, j] = 0;
                    }
                    if (i == j)
                    {
                        L[i, j] = 1;
                    }
                    if (i <= j)
                    {
            U[i, j] = Source_Matrix[i, j];
                    }
                }
            }
            if (null != done) done();
        }
```

Using the LU functions above provides an informative exercise, but explicitly forming individual matrices L and U is usually not necessary. The Gaussian elimination procedures when performed on a square matrix return an encoded LU result. The upper triangle is U and the elements below the main diagonal (the row factors) are the elements below the unit diagonal of L. We will soon see that solutions can be obtained directly on this encoded LU and the effort and memory waste of the parsing algorithms can be eliminated.

Permutation Matrices and Encoded Permutation Matrices

Up until now I have been using the encoded column and row permutation matrices in the procedures without explanation, other than to state that they record exchanges. A permutation matrix begins as an identity matrix. Any time a row (column) exchange occurs on our transforming matrix, that exchange is also performed on the permutation matrix. A permutation matrix is square. Consider a three by three case where columns 1 and 3 are exchanged, then columns 2 and 3 are exchanged. The column permutation matrix transforms as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

We encode the initial permutation matrix as a vector {1, 2, 3}. As exchanges in the matrix columns occur, the corresponding elements of the encoded permutation matrix are exchanged. We get {1, 2, 3} → {3, 1, 2} for the above sequence of exchanges.

The same logic applies to row permutation matrices. We must maintain separate permutation matrices for column exchanges and row exchanges in either the explicit or encoded case.

An encoded permutation matrix conserves memory and reduces computational overhead. We do not have to store all of those zeros when we encode. What is more profound, we do not have to multiply an *mxm* matrix by an *mxm* matrix. To effect a multiplication by a column permutation matrix stored in encoded form, we simply need to swap columns in both the matrix and the encoded column permutation matrix (performing exactly the same operations on the matrix's columns as the permutation vector's elements) as necessary to return the permutation to 1, 2, 3,… order.

Likewise, to effect a multiplication on a row permutation matrix stored in encoded form by a matrix, we swap rows in the matrix and the elements of the encoded row permutation matrix (again performing exactly the same operations on the matrix's rows as the permutation vector's elements) as necessary to return the permutation to 1, 2, 3,… order. We will see this strategy implemented later. For now it is important to understand the general concept.

Using explicit permutation matrices is an exquisite waste of processor time and an unnecessary waste of memory, but there may be times one actually wants a full blown permutation matrix. (They are useful in learning.) The following two functions decode encoded permutations:

```
        public double[,] Decode_column_permutation(int[]
encoded_vector)
        {
            int n = encoded_vector.GetLength(0) - 1;
            double[,] Cperm = new double[n + 1, n + 1];
            double[,] temp = new double[n + 1, n + 1];
```

```
        int i = 0, j = 0;
        for (i = 1; i <= n; i++)
        {
            temp[i, i] = 1;
        }
        for (i = 1; i <= n; i++)
        {
            for (j = 1; j <= n; j++)
            {
                Cperm[i, j] = temp[i, encoded_vector[j]];
            }
        }
        return Cperm;
    }

    public double[,] Decode_row_permutation(int[] encoded_vector)
    {
        int n = encoded_vector.GetLength(0) - 1;
        double[,] Rperm = new double[n + 1, n + 1];
        double[,] temp = new double[n + 1, n + 1];
        int i = 0, j = 0;
        for (i = 1; i <= n; i++)
        {
            temp[i, i] = 1;
        }
        for (i = 1; i <= n; i++)
        {
            for (j = 1; j <= n; j++)
            {
                Rperm[i, j] = temp[encoded_vector[i], j];
            }
        }
        return Rperm;
          }
```

Determined Systems

He we consider the system $Ax = b$ where $A \in \mathbb{R}^{mn}$, x is an n dimensional vector, b is an m dimensional vector, and $m = n$. This is the standard problem of solving a system of linear equations where the number of equations is equal to the number of unknowns. Remembering that $AA^{-1} = A^{-1}A = I$, let us multiply both sides of the equation $Ax = b$ by A^{-1}: $x = A^{-1}b$. So, the solution is simple; all we have to do is multiply the b vector by A^{-1}. However, inverting a matrix is computationally expensive.

Another overwhelmingly expensive procedure with historical significance is Cramer's rule:

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

$$x_1 = \frac{\det\begin{pmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{pmatrix}}{\det\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}},\ x_2 = \frac{\det\begin{pmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{pmatrix}}{\det\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}}, \text{ etc.}$$

A General Practical Solution

Triangular systems are easy to solve. For example, consider the following m=3 determined system:

$$\begin{bmatrix} 1 & 3 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & -4 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ -1 \\ -12 \end{bmatrix}$$

$x_3 = -12/-4 = 3,\ x_2 = (-1-(-1*3)/1 = 2$, etc.

The process we use is called back calculation. It is encapsulated in the following code:

```
b(n, 1) = b(n, 1) / A(n, n)
For i = n - 1 To 1 Step -1
   For j = n To i + 1 Step -1
      temp = temp - A(i, j) * b(j, 1)
   Next
   temp = temp + b(i, 1)
   b(i, 1) = temp / A(i, i)
   temp = 0
Next
x = b
```

Similarly, a lower triangular system can be solved with forward calculation:

```
b(1, 1) = b(1, 1) / A(1, 1)
For i = 2 To n
   For j = 1 To i - 1
      Temp = Temp - A(i, j) * b(j, 1)
   Next
   Temp = Temp + b(i, 1)
```

```
    b(i, 1) = Temp / A(i, i)
    Temp = 0
  Next
  x = b
```

The trouble is, most systems do not have a triangular coefficient matrix, but the LU factorization will resolve any non-singular mxm matrix into triangular factors L and U. Obtaining a solution via LU factorization is perhaps the best generally applicable approach. Recall that $L, U, P, and\ Q$ make up the LU decomposition and $PAQ = LU$.

To use the full pivot LU factorization to solve $Ax = b$ for x:

Obtain L, U, P and Q for the coefficient matrix A
Solve for y using the formula Ly=Pb
Solve for z using the formula Uz=y
Multiply z by Q (Qz) to obtain x

If a row pivot Gaussian elimination was used, then $PA = LU$ because Q is the identity matrix.

Obtain L, U, and P for the coefficient matrix A
Solve for y using the formula Ly=Pb
Solve for x using the formula Ux=y

You may wonder what the significance of y is in the above algorithms. It is the transformed b that would be obtained if it were subjected to the same row operations performed on A in the LU factorization.

Note that once L, U, P and Q have been determined, these factors can be reused to solve an infinity of systems provided the coefficient matrix A remains the same. It is of substantial importance that the LU factorization is *reusable* in this way.

Below is a solution function that takes L, U, b, and optionally P and Q as arguments and returns x. P must be included if row pivoting was used. P and Q must be included if full pivoting was used.

```
        public double[,] Explicit_LU_Ax_b(double[,] L, double[,] U,
double[,] b, int[] Gauss_Encoded_P, int[] Gauss_Encoded_q)
        {
            // A preliminary LU Factorization is required
            // Include optional Row_Permutation_Matrix if any pivot
strategy was used
            // Include optional Column_Permutation_Matrix if full pivot
LU is used
            if (L.GetLength(0) != L.GetLength(1))
            {
                throw new ApplicationException("matrix must be
square");
            }
            int n = L.GetLength(0) - 1;
            double[,] localb = new double[n + 1, 2];
```

```
            Array.Copy(b, localb, b.Length);
            int i = 0, j = 0;
            if (Gauss_Encoded_P == null)
            {
                // take no action pb = localb
            }
            else
            {
                double[,] tempb = new double[n + 1, 2]; //  multiply by
permutation
                for (i = 1; i <= n; i++)
                {
                    tempb[i, 1] = localb[Gauss_Encoded_P[i], 1];
                }
                localb = tempb;
            }
            double temp = 0;
            // forward calculate for Ly=Pb
            for (i = 2; i <= n; i++)
            {
                for (j = 1; j <= i - 1; j++)
                {
                    temp = temp - L[i, j] * localb[j, 1];
                }
                localb[i, 1] = temp + localb[i, 1];
                temp = 0;
            }
            // At this point localb holds y. We will back calculate for
Uz=y
            localb[n, 1] = localb[n, 1] / U[n, n];
            for (i = n - 1; i >= 1; i += -1)
            {
                for (j = n; j >= i + 1; j += -1)
                {
                    temp = temp - U[i, j] * localb[j, 1];
                }
                temp = temp + localb[i, 1];
                localb[i, 1] = temp / U[i, i];
                temp = 0;
            }
            // At this point localb holds z
            if (Gauss_Encoded_q == null)
            {
                if (null != done) done();
                return localb;
            }
            else
            { // Multiply by permutation matrix

                double[,] tempb = new double[n + 1, 2];
                for (i = 1; i <= n; i++)
                {
                    tempb[Gauss_Encoded_q[i], 1] = localb[i, 1];
                }
                localb = tempb;
                if (null != done) done();
                return localb;
```

```
            }
        }

        public double[,] Explicit_LU_Ax_b(double[,] L, double[,] U,
double[,] b)
        {
            return Explicit_LU_Ax_b(L, U, b, null);
        }

        public double[,] Explicit_LU_Ax_b(double[,] L, double[,] U,
double[,] b, int[] Gauss_Encoded_P)
        {
            return Explicit_LU_Ax_b(L, U, b, Gauss_Encoded_P, null);
              }
```

Again, explicit formation of L and U is unnecessary. The following function takes the result of the Gaussian elimination function (designated LU_encoded), as well as b and optionally P and Q as arguments, and returns x.

```
        public double[,] Encoded_LU_Ax_b(double[,] LU_encoded,
double[,] b, int[] Gauss_Encoded_P, int[] Gauss_Encoded_q)
        {
            // LU solution from encoded LU
            // Include optional Row_Permutation_Matrix if any pivot
strategy was used
            // Include optional Column_Permutation_Matrix if full pivot
LU is used
            int i = 0, j = 0;
            int m = LU_encoded.GetLength(0) - 1;
            int n = LU_encoded.GetLength(1) - 1;
            double[,] localb = new double[n + 1, 2];
            Array.Copy(b, localb, localb.Length);
            if (m != n)
            {
                throw new ApplicationException("matrix must be
square");
            }
            if (Gauss_Encoded_P == null)
            {
                // take no action
            }
            else
            {
                double[,] tempb = new double[n + 1, 2];
                for (i = 1; i <= n; i++)
                {
                    tempb[i, 1] = localb[Gauss_Encoded_P[i], 1];
                }
                localb = tempb; // localb now holds pb
            }
            double temp = 0;
            // forward calculate for Ly=Pb
            for (i = 2; i <= n; i++)
            {
                for (j = 1; j <= i - 1; j++)
                {
```

```
                    temp = temp - LU_encoded[i, j] * localb[j, 1];
                }
                localb[i, 1] = temp + localb[i, 1];
                temp = 0;
            }
            // localb now holds y
            // back calculate for Uz=y
            localb[n, 1] = localb[n, 1] / LU_encoded[n, n];
            for (i = n - 1; i >= 1; i += -1)
            {
                for (j = n; j >= i + 1; j += -1)
                {
                    temp = temp - LU_encoded[i, j] * localb[j, 1];
                }
                temp = temp + localb[i, 1];
                localb[i, 1] = temp / LU_encoded[i, i];
                temp = 0;
            }
            // localb now holds z = un permuted x
            if (Gauss_Encoded_q == null)
            {
                if (null != done) done();
                return localb;
            }
            else
            {
                double[,] tempb = new double[n + 1, 2];
                for (i = 1; i <= n; i++)
                {
                    tempb[Gauss_Encoded_q[i], 1] = localb[i, 1];
                }
                localb = tempb;
                if (null != done) done();
                return localb;
            }
        }
        public double[,] Encoded_LU_Ax_b(double[,] LU_encoded,
double[,] b)
        {
            return Encoded_LU_Ax_b(LU_encoded, b, null);
        }

        public double[,] Encoded_LU_Ax_b(double[,] LU_encoded,
double[,] b, int[] Gauss_Encoded_P)
        {

            return Encoded_LU_Ax_b(LU_encoded, b, Gauss_Encoded_P,
null);
              }
```

As a final note on the solution of determined systems when using Gaussian elimination, I must mention that the algorithms depending on it sometimes fail due to catastrophic cancellation, scaling problems, and certain other potential algorithmic instabilities. Chapter 6 discusses some of these problems and how to detect them. Golub and Watkins provide extensive detail.

An example determined system:

$0.17x_1 + 11.4x_2 + 5.91x_3 = 19.1$
$1.63x_1 + 11.7x_2 + 6.61x_3 = 11.75$
$3.11x_1 + 6.00x_2 + 7.31x_3 = 4.23$

The coefficient matrix is: $A = \begin{bmatrix} 0.17 & 11.4 & 5.91 \\ 1.63 & 11.7 & 6.61 \\ 3.11 & 6.00 & 7.31 \end{bmatrix}$

The response vector is: $b = \{19.1 \quad 11.75 \quad 4.23\}^T$

Using a full pivot strategy, the encoded LU factorization is:

$$LU = \begin{bmatrix} 11.7 & 6.61 & 1.63 \\ 0.513 & 3.920 & 2.274 \\ 0.974 & -0.135 & -1.110 \end{bmatrix}$$

We can parse this encoded LU into:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.513 & 1 & 0 \\ 0.974 & -0.135 & 1 \end{bmatrix} \text{ and } U = \begin{bmatrix} 11.7 & 6.61 & 1.63 \\ 0 & 3.920 & 2.274 \\ 0 & 0 & -1.110 \end{bmatrix}$$

The permutation matrices are:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \text{ and } Q = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The result is: $x = \{-6.671 \quad 0.006095 \quad 3.412\}^T$

The Matrix Inverse Revisited

Now that we have a stable method for efficiently computing a series of solutions for Ax=b where the A matrix remains a constant array, let us see how we can implement a better approach for calculating A^{-1}.

Remember $AA^{-1} = I$. Clearly, this is a "matrix × matrix = matrix" problem and not the "matrix × vector = vector" type of problem we just reviewed. However, it can be split into a series of LU solutions by considering $A\{A_1^{-1}...A_m^{-1}\} = \{I_1...I_m\}$, where A_k^{-1} and I_k are the k^{th} columns of A^{-1} and I, respectively. We simply have to solve them one at a time, i.e., $AA_1^{-1} = I_1$, $AA_2^{-1} = I_2$, etc. This isn't as much work as you might think, because we only have to do the LU factorization once since it is reusable with a constant coefficient matrix. Here is the encapsulation of this abstraction:

```
        public double[,] Inverse_by_LU_encoded(double[,] Source_Matrix)
        {
            // Watkins p 103 Overwrites Source_Matrix encoded LU
            if (Source_Matrix.GetLength(0) !=
Source_Matrix.GetLength(1))
            {
                throw new ApplicationException("matrix must be
square");
            }
            int m = Source_Matrix.GetLength(0) - 1;
            double[,] result = new double[m + 1, m + 1];
            int i = 0, j = 0;
            Source_Matrix = GaussUT_full_piv(Source_Matrix);
            int[,] identity = new int[m + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                identity[i, i] = 1;
            }
            double[,] temp1 = new double[m + 1, 2];
            double[,] temp2 = new double[m + 1, 2];
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= m; j++)
                {
                    temp1[j, 1] = identity[i, j];
                }
                temp2 = Encoded_LU_Ax_b(Source_Matrix, temp1,
Gauss_Encoded_P, Gauss_Encoded_Q); // This function uses global
permutation matrices.
                for (j = 1; j <= m; j++)
                {
                    result[j, i] = temp2[j, 1];
                }
            }
            if (null != done) done();
            return result;
        }
```

Symmetric Positive Definite Systems, The Cholesky Factorization

So far our algorithms for determined systems are universally applicable. However, they are not necessarily the least computationally expensive approaches for certain special case systems. One special case is the symmetric positive definite system.

Linear systems involving symmetric positive definite coefficient matrices can be solved with a Cholesky factorization, which requires $m^3/3$ FLOPs as opposed to $2m^3/3$ FLOPs for an LU factorization.

Consider a square matrix with dimensions mxm and the set of all m dimensional vectors. Remember that vectors are matrices with one row (or column). The product $x^T Ax$ will be a scalar. If this number is greater than zero, no matter what m dimensional vector x we consider (except the case that x is a zero vector), then the matrix A is said to be *positive definite*; i.e., a matrix $A \in \mathbb{R}^{m,m}$ is positive definite if $x^T Ax > 0$ for all non zero $x \in \mathbb{R}^m$.

If, for $A \in \mathbb{R}^{m,m}$, $A^T = A$, then A is said to be symmetric.

If $A \in \mathbb{R}^{m,m}$ is symmetric positive definite, then there exists a unique lower triangular matrix $G \in \mathbb{R}^m$ with positive diagonal elements such that $GG^T = A$. GG^T is the Cholesky factorization of A.

First, here is a function for obtaining G:

```
public double[,] Cholesky(double[,] Source_Matrix)
{
    // cholesky gaxpy Golub/Van Loan 4.2.1 Result = G
    // Overwrites Source_Matrix and zeros above diagonal
    int i = 0, j = 0, k = 0;
    int n = Source_Matrix.GetLength(0) - 1;
    double[] v = new double[n + 1];
    for (j = 1; j <= n; j++)
    {
        for (i = j; i <= n; i++)
        {
            v[i] = Source_Matrix[i, j];
        }
        for (k = 1; k <= j - 1; k++)
        {
            for (i = j; i <= n; i++)
            {
                v[i] = v[i] - (Source_Matrix[j, k] *
Source_Matrix[i, k]);
            }
        }
        for (i = 1; i <= n; i++)
        {
            if (i < j)
            {
                Source_Matrix[i, j] = 0;
            }
            else
            {
                Source_Matrix[i, j] = v[i] / (Math.Pow(v[j],
0.5));
            }
        }
    }
    if (null != done) done();
    return Source_Matrix;
```

```
}
```

One good way to determine if your matrix is positive definite is to submit it to the above function. If the code executes and does not return negative square roots (NaN), the matrix submitted is positive definite. Its symmetry can be evaluated by inspection, or by verifying that its product with the inverse of its transpose yields the identity matrix.

Once we have a Cholesky factorization of A we can obtain solutions. Given a determined system $Ax = b$ where A is symmetric positive definite, we can use the following steps to solve for x:

Obtain G and G^T
Solve $Gy = b$ (simple forward calculation)
Solve $G^T x = y$ (simple back calculation)

This algorithm is encapsulated below:

```
public double[,] Cholesky_Ax_b(double[,] G, double[,] b)
{
    if (G.GetLength(0) != G.GetLength(1))
    {
        throw new ApplicationException("matrix must be
square");
    }
    int n = G.GetLength(0) - 1;
    double[,] result = new double[n + 1, 2];
    Array.Copy(b, result, b.Length); // result now holds b
    int i = 0, j = 0;
    double Temp = 0;
    // forward calculate for Gy=b
    result[1, 1] = result[1, 1] / G[1, 1];
    for (i = 2; i <= n; i++)
    {
        for (j = 1; j <= i - 1; j++)
        {
            Temp = Temp - G[i, j] * result[j, 1];
        }
        Temp = Temp + result[i, 1];
        result[i, 1] = Temp / G[i, i];
        Temp = 0;
    }
    // result now holds y
    double[,] GT = new double[n + 1, n + 1];
    GT = Transpose(G);
    Temp = 0;
    // back calculate for Ux=y
    result[n, 1] = result[n, 1] / GT[n, n];
    for (i = n - 1; i >= 1; i += -1)
    {
        for (j = n; j >= i + 1; j += -1)
        {
            Temp = Temp - GT[i, j] * result[j, 1];
        }
        Temp = Temp + result[i, 1];
```

```
            result[i, 1] = Temp / GT[i, i];
            Temp = 0;
        }
        // result now holds x
        if (null != done) done();
        return result;
    }
```

3. The QR Factorization and Full Rank, Over Determined Systems

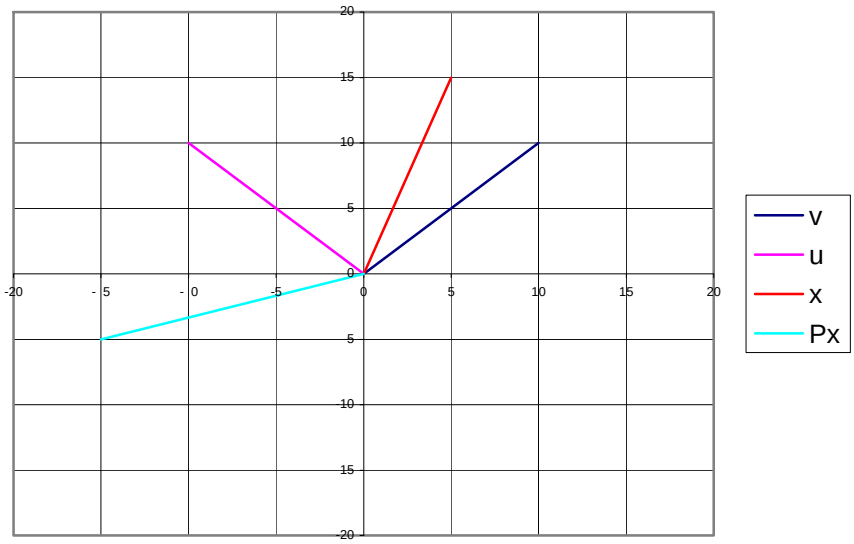
In the last chapter we discussed Gaussian elimination and saw how it could be applied to matrices where $m \leq n$. The LU factorization was simply Gaussian elimination applied to square matrices. The LU factorization was found to be of general use in solving determined systems.

In this chapter we will discuss the QR factorization as it applies to matrices where $m \geq n$. There are several variants we will discuss. The QR factorization is of general use in solving over determined systems. Here there will be more equations than unknowns, and we will seek the *best* solution set.

The QR Factorization

QR Factorization by Householder Reflection

Consider the following two dimensional system:



We have a vector v. Perpendicular to it is the vector u. Another vector x has a reflection through u designated Px. Suppose we choose v such that Px lies exactly on the x axis. Px will have a zero y coefficient but will have the same magnitude (the same Euclidian norm) as x. This is a direct artifact of the Pythagorean theorem.

We can extend the model to three dimensions, reflecting x through a plane u which is perpendicular to a plane v. Finally, in m-dimensional space, we can obtain a

hyperplane Px by reflecting x through a hyperplane u which is orthogonal to a hyperplane v. The effect, regardless of m, is that if we choose v properly we can selectively zero elements of x and obtain a new vector Px which has the same Euclidian norm as x.

When the operator P is chosen to strategically zero elements in a vector, it is referred to as the *Householder matrix.* The Householder matrix's action on x is a matrix–vector multiplication. The Householder matrix is typically designated H and it is obtained as follows: $H = I_{m,m} - \frac{2}{v^T v} vv^T$, where we consider v a column vector. Note that $-\frac{2}{v^T v}$ is a scalar and vv^T is an mxm matrix. When v is chosen in such a fashion that Hx (also a column vector) has zeros below the first row, then v is called the *Householder vector*. We only care about its direction; its magnitude is not a concern, so we normalize the vector in such a fashion that its first dimension is unity. It is computed with the following algorithm:

```
        public double[] House(double[] x_vector)
        {
            // Golub/Van Loan 5.1.1
            int i = 0;
            int m = x_vector.Length - 1;
            // scale to prevent overflow
            double N2 = 0;
            for (i = 1; i <= m; i++)
            {
                N2 = Math.Pow(x_vector[i], 2) + N2;
            }
            N2 = Math.Pow(N2, 0.5);
            for (i = 1; i <= m; i++)
            {
                x_vector[i] = x_vector[i] / N2;
            }
            //procede with calculation
            double u = 0;
            double sigma = 0;
            double[] Result = new double[m + 1];
            for (i = 2; i <= m; i++)
            {
                sigma = sigma + Math.Pow(x_vector[i], 2);
                Result[i] = x_vector[i];
            }
            if (sigma == 0)
            {
                HOUSE_BETA = 0;
            }
            else
            {
                u = Math.Pow(((Math.Pow(x_vector[1], 2)) + sigma),
0.5);
                if (x_vector[1] <= 0)
                {
                    Result[1] = x_vector[1] - u;
```

```
                }
                else
                {
                    Result[1] = -sigma / (x_vector[1] + u);
                }
                HOUSE_BETA = (2 * Result[1] * Result[1]) / (sigma +
(Math.Pow(Result[1], 2)));
                for (i = 2; i <= m; i++)
                {
                    Result[i] = Result[i] / Result[1];
                }
            }
            Result[1] = 1;
            if (null != done) done();
            return Result;
        }
```

An example:

$x = (1,3,5)^T$

$v = (1,-0.610242,-1.0170706)^T$

$Hx = (5.9160788,0,0)^{\mathrm{T}}$

Our next preliminary concept is that of an orthogonal matrix. Remember that orthogonal vectors are, in a functional sense, perpendicular, and orthonormal vectors are orthogonal vectors with length equal to unity. A matrix composed of orthonormal column (row) vectors is an orthogonal matrix. Of utmost importance is the fact that if Q is orthogonal, then $Q^T = Q^{-1}$. The columns (rows) of an orthogonal matrix can form an orthonormal basis for vectors.

The Householder matrix is an orthogonal matrix. The column (row) vectors comprising the Householder matrix each have a magnitude of unity (they are orthonormal vectors). Any vector we multiply by any of these vectors will change its direction, but not its magnitude. When a matrix is multiplied by an orthogonal matrix, the magnitudes and angles between the vectors of the matrix are preserved. The overall orientation of the matrix in hyperspace is changed. (This commonly used concept of redirecting vectors is unnecessary, and later we will see that we are in fact simply representing the vectors in a more convenient basis.)

Remember that upper triangular systems are easy to solve. We seek next to identify a way of performing this transformation to factor a matrix so as to yield an upper triangular *nxn* matrix R and its associated new *mxm* orthonormal basis Q.

For A^{mxn} where $m \geq n$, there is an orthogonal matrix Q^{mxm} and an upper triangular matrix R^{nxn} such that $A = QR$. Let us consider an example:

$$A = \begin{pmatrix} 1.5 & 1.84 & 2.76 \\ 2.5 & 3.95 & 9.88 \\ 3.5 & 6.55 & 22.92 \\ 4.5 & 9.55 & 42.96 \end{pmatrix}.$$

We submit the first column to the House function and obtain:

$$v = \begin{pmatrix} 1 \\ -0.50988 \\ -0.71383 \\ -0.91778 \end{pmatrix}.$$

The Householder matrix is:

$$H_1 = I_{m,m} - \frac{2}{v^T v} v v^T = \begin{pmatrix} 0.2342606 & 0.3904344 & 0.5466082 & 0.7027819 \\ 0.3904344 & 0.8009257 & -0.278704 & -0.3583337 \\ 0.5466082 & -0.278704 & 0.6098144 & -0.5016672 \\ 0.7027819 & -0.3583337 & -0.5016672 & 0.3549993 \end{pmatrix}.$$

The product of the Householder matrix and the original matrix is:

$$A_1 = \begin{pmatrix} 6.403124 & 12.26511 & 47.22382 \\ 0 & -1.365543 & -12.79117 \\ 0 & -0.891759 & -8.819633 \\ 0 & -0.017977 & 2.151897 \end{pmatrix}.$$

We next submit the last three rows of column two of A_1 to the House function and obtain $v = \begin{pmatrix} 1 \\ 0.2975928 \\ 0.0059991 \end{pmatrix}$. To this vector we must append a leading zero to obtain $v = \begin{pmatrix} 0 \\ 1 \\ 0.2975928 \\ 0.0059991 \end{pmatrix}$ The Householder matrix is:

$$H_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.8372263 & -0.5467454 & -0.01102165 \\ 0 & -0.5467454 & 0.8372925 & -0.00327996 \\ 0 & -0.0110217 & -0.0032800 & 0.9999339 \end{pmatrix}.$$

The product of this Householder matrix and A_1 is:

$$A_2 = \begin{pmatrix} 6.403124 & 12.26511 & 47.22382 \\ 0 & 1.631032 & 15.50748 \\ 0 & 0 & -0.3981584 \\ 0 & 0 & 2.321662 \end{pmatrix}.$$

Finally, the last two rows of A_2 give $v = \begin{pmatrix} 1 \\ -0.8431019 \end{pmatrix}$. We add leading zeros to obtain $v = \begin{pmatrix} 0 \\ 0 \\ 1 \\ -0.8431019 \end{pmatrix}$.

The corresponding Householder is:

$$H_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -0.1690295 & 0.985611 \\ 0 & 0 & 0.985611 & 0.1690295 \end{pmatrix}.$$

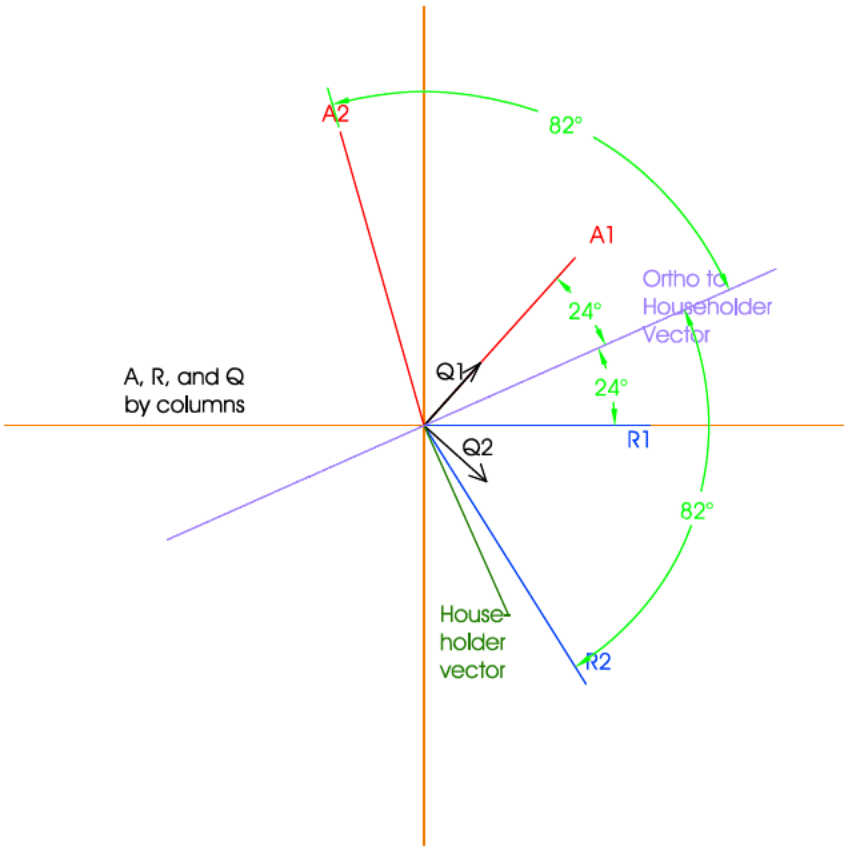
This Householder times A_2 gives

$$A_3 = \begin{pmatrix} 6.403124 & 12.26511 & 47.22382 \\ 0 & 1.631032 & 15.50748 \\ 0 & 0 & 2.355556 \\ 0 & 0 & 0 \end{pmatrix}.$$

Qualitatively what have we done? We have realigned the n column vectors of A in m-space such that only the first n dimensions are required to describe the magnitude and direction of those vectors. We did this in such a way as to enable those vectors to be progressively designated with 1, 2 …n dimensions. We have also maintained a record of the transformations needed to perform this realignment: H_1, H_2 and H_3. Forming R begins with $H_1 A$. The next step is $H_2(H_1 A)$ and so on resulting in $R = \left(\prod_{k=n}^{1} H_k \right) A$. The term $\left(\prod_{k=n}^{1} H_k \right)$ results in an orthogonal matrix. We define Q as its transpose:

$Q = \left(\prod_{k=n}^{1} H_k\right)^T = \left(\prod_{k=n}^{1} H_k\right)^{-1} = \left(\prod_{k=1}^{n} H_k\right)$. Qualitatively, Q represents a multiplicative "key" that can operate on our transformed matrix R, undoing those transformations and returning it to its original state A, i.e., $A = QR$.

As promised, there is an alternative understanding of this process. I do not like the idea of vectors having their direction changed; rather, I prefer to see the process as a change of basis (or a re-envisioning in a new reference frame). Recall how in Chapter One we examined basis in the Introductory Vector and Matrix Terminology section. Let's see how these two ideas look together in one diagram. Below you can see in a two dimensional Cartesian system two vectors (A1 and A2) of a 2x2 matrix together with their reflections (R1 and R2) through a line ortho to the Householder vector for A1. You can also see the two vectors of the Q matrix. If we represent A1 and A2 in a new reference frame defined by the axis Q1 and Q2, we will obtain the very same coordinates that we have for R1 and R2 in the original reference frame. So, we don't really need new vectors, just a new basis (selected very carefully) and a re-expression of A1 and A2. That new basis is exactly what Q is.



In practice we never really form Householder matrices. We know that $HA = (I - \beta vv^T)A$. (where $\beta = \frac{2}{v^T v}$) We can multiply out the righthand side and obtain

$A - \beta v v^T A$. Recall that $AB = \left(B^T A^T\right)^T$ and obtain $HA = A - \beta v \left(A^T v\right)^T$. Similarly, $AH = A(I - \beta v v^T) = A - \beta A v v^T$.

When we employ this approach we compute matrix–vector multiplications instead of costly matrix–matrix multiplications.

The following code encapsulates the basic QR decomposition abstraction:

```
        public double[,] Householder_QR_Simple(double[,] Source_Matrix)
        {
            // Golub/Van Loan 5.2.1 Overwrites source_matrix with R and
returns Q from function call.
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            QRRank = n; // Default for full rank case.
            int i = 0, j = 0, k = 0;
            double[,] Q = new double[m + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                Q[i, i] = 1;
            }
            double[] vector = new double[m + 1];
            double[,] v = new double[m + 1, 2];
            beta = new double[n + 1];
            for (j = 1; j <= n; j++)
            {
                v = new double[m + 1, 2];
                vector = new double[m - j + 2];
                // convert the jth column of the submatrix to a 1
dimensional vector for submission to house
                for (i = 1; i <= vector.Length - 1; i++)
                {
                    vector[i] = Source_Matrix[j + i - 1, j];
                }
                // submit it
                vector = House(vector);
                // convert the result to a 2 dimensional "vector" with
leading zeros up to the jth row
                for (i = 1; i <= vector.Length - 1; i++)
                {
                    v[j + i - 1, 1] = vector[i];
                }
                beta[j] = HOUSE_BETA;
                double[,] omegaT = null;
                omegaT = Matrix_Multiply(v,
Scalar_multiply(Transpose(Matrix_Multiply(Transpose(Source_Matrix),
v)), beta[j]));
                for (i = j; i <= m; i++)
                {
                    for (k = j; k <= n; k++)
                    {
                        Source_Matrix[i, k] = Source_Matrix[i, k] -
omegaT[i, k];
                    }
```

```
                }
                double[,] omega = null;
                omega =
Scalar_multiply(Matrix_Multiply(Matrix_Multiply(Q, v), Transpose(v)),
beta[j]);
                for (i = 1; i <= m; i++)
                {
                    for (k = 1; k <= m; k++)
                    {
                        Q[i, k] = Q[i, k] - omega[i, k];
                    }
                }
            }
            if (null != done) done();
            return Q;
        }
```

Note that when we form omega above, we perform the multiplication $(Qv)v^T$ and not $Q(vv^T)$. Although they give the same result, the former is considerably less computationally expensive.

Many applications do not require the explicit generation of the Q matrix, so an encoded array can be used where the upper triangle is the R matrix and the elements below the principal diagonal represent the elements of a unit lower triangular matrix of the Householder vectors. The following procedure implements these modifications:

```
        public double[,] Householder_QR_Simple_encoded(double[,]
Source_Matrix)
        {
            // QR Golub/Van Loan 5.2.1 Form encoded QR without explicit
formation of Householder matrices
            // Overwrites Source_matrix with encoded QR and also sends
encoded QR as return
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            QRRank = n; // Default for full rank case.
            int i = 0, j = 0, k = 0;
            beta = new double[n + 1];
            for (j = 1; j <= n; j++)
            {
                double[,] v = new double[m + 1, 2]; // redim is used
simply to reinitialize the array here
                double[] vector = new double[m - j + 1 + 1];
                // convert the jth column of the submatrix to a 1
dimensional vector for submission to house
                for (i = 1; i <= vector.Length - 1; i++)
                {
                    vector[i] = Source_Matrix[j + i - 1, j];
                }
                // submit it
                vector = House(vector);
                // convert the result to a 2 dimensional "vector" with
leading zeros up to the jth row
                for (i = 1; i <= vector.Length - 1; i++)
                {
```

```
                    v[j + i - 1, 1] = vector[i];
                }
                beta[j] = HOUSE_BETA;
                double[,] omegaT = null;
                omegaT = Matrix_Multiply(v,
Scalar_multiply(Transpose(Matrix_Multiply(Transpose(Source_Matrix),
v)), beta[j]));
                for (i = j; i <= m; i++)
                {
                    for (k = j; k <= n; k++)
                    {
                        Source_Matrix[i, k] = Source_Matrix[i, k] -
omegaT[i, k];
                    }
                }
                // populate the lower triangular matrix holding the
Householder vectors
                for (i = j + 1; i <= m; i++)
                {
                    Source_Matrix[i, j] = vector[i - j + 1];
                }
            }
            if (null != done) done();
            return Source_Matrix;
        }
```

The encoded result can be decoded into a discrete Q and R result with the following decoder subroutine:

```
        public double[,] Decode_Householder_QR(double[,]
Encoded_matrix)
        {
            // Returns Q from encoded QR, R is just the upper triangle
of encoded_R.
            // Conserves Encoded_R
            int m = Encoded_matrix.GetLength(0) - 1;
            int n = Encoded_matrix.GetLength(1) - 1;
            int i = 0, j = 0, k = 0;
            double[,] Q = new double[m + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                Q[i, i] = 1;
            }
            // Decoder for essential part of Householder vectors
storred in lower triangle.
            for (j = n; j >= 1; j += -1)
            {
                double[,] v = new double[m + 1, 2];
                v[j, 1] = 1;
                for (i = j + 1; i <= m; i++)
                {
                    v[i, 1] = Encoded_matrix[i, j];
                }
                double[,] omega = null;
                omega = Matrix_Multiply(Transpose(Q), v);
                for (i = 1; i <= m; i++)
```

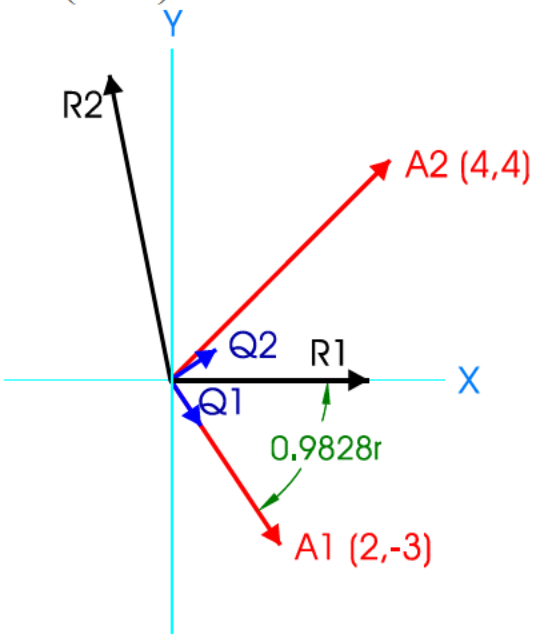
```
                {
                    for (k = 1; k <= m; k++)
                    {
                        Q[i, k] = Q[i, k] - 1 * beta[j] * omega[k, 1] *
v[i, 1]; // This beta(j) is a global
                    }
                }
            }
            return Q;
        }
```

Be aware of the global beta. It is changed at class level each time a QR is executed. If you might need it after an interceding QR, you should copy it into another array in your interface code.

QR Factorization by Givens (Jacobi) Rotations

Consider the following system:

$$A = \begin{pmatrix} 2 & 4 \\ -3 & 4 \end{pmatrix}$$



Our objective is to rotate the column vectors of A counterclockwise to zero the lower left element of A, creating an upper triangular matrix R. Clearly we need to rotate the A matrix 0.9828 radians in the xy plane. This we can accomplish by multiplying the A matrix by an orthogonal matrix $Q^T = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$. Again, we may understand this transformation as a re-expression of A in terms of a new basis Q, eliminating the concept of re-directing vectors.

In n dimensions we can rotate a matrix in the (i,k) coordinate plane by multiplying by a modified identity matrix $G(i,k,\theta)^T$, where

$$G(i,k,\theta)=\begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \cos(\theta) & \sin(\theta) & 0 & \ddots & \vdots \\ \vdots & \ddots & -\sin(\theta) & \cos(\theta) & 0 & \ddots & \vdots \\ \vdots & \ddots & 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{pmatrix} \begin{matrix} \\ \\ i \\ k \\ \\ \\ \\ \end{matrix}$$

$$\begin{matrix} & & i & k & & & \end{matrix}$$

How do we determine θ such that the element $a_{k,i}$ is zeroed? Or, to restate the question, what must θ be to effect the following transformation?

$$\begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}^T \begin{pmatrix} a_{ii} \\ a_{ki} \end{pmatrix} = \begin{pmatrix} r_{ii} \\ 0 \end{pmatrix}$$

The following function returns a two dimensional array "result" that holds cos(θ) in result(1) and sin(θ) in result(2):

```
        public double[] Givens(double a, double b)
        {
            // Golub/Van Loan 5.1.3
            // (c,s)=givins(a,b): the cosine is returned in Result(1),
the sine in Result(2)
            double[] Result = new double[4];
            double tau = 0;
            double c = 0;
            double s = 0;
            if (b == 0)
            {
                c = 1;
                s = 0;
            }
            else
            {
                if (System.Math.Abs(b) > System.Math.Abs(a))
                {
                    tau = -a / b;
                    s = 1 / (Math.Pow((1 + (Math.Pow(tau, 2))), 0.5));
                    c = s * tau;
                }
                else
                {
                    tau = -b / a;
                    c = 1 / (Math.Pow((1 + (Math.Pow(tau, 2))), 0.5));
                    s = c * tau;
                }
            }
```

```
            Result[1] = c;
            Result[2] = s;
            return Result;
        }
```

The Givens function enables us to selectively zero elements anywhere in a matrix. We can use this function to zero all of the elements below the principal diagonal of a matrix to obtain an upper triangular matrix R. We can accumulate the orthogonal transformation matrices $G(i,k,\theta) = Q$. This process will yield the decomposition $A = QR$ (or $Q^T A = R$). Unlike the Householder approach, elements in A are zeroed one at a time.

Performing the multiplication $A = G(i,k,\theta)^T A$ can be made substantially less costly by recognizing that $G(i,k,\theta)^T$ is — for m > 2 — mostly an identity matrix. Just the i[th] and k[th] rows of A are changed. Likewise, $A = AG(i,k,\theta)$ changes just the i[th] and k[th] columns of A.

The following subroutine performs the decomposition:

```
        public double[,] Givens_QR_Simple(double[,] Source_Matrix)
        {
            // Golub/Van Loan 5.2.2 Overwrites Source_Matrix with R
returns Q
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            QRRank = n; // by default
            int h = 0, i = 0, j = 0;
            double[] cs = new double[4];
            double[,] Q = new double[m + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                Q[i, i] = 1;
            }
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(Source_Matrix[i, j]) >
Givens_Zero_Value)//  If it's already zero we won't zero it.
                    {
                        double[,] temp_R = new double[3, n - j + 1 +
1];
                        double[,] temp_Q = new double[m + 1, 3];
                        cs = Givens(Source_Matrix[i - 1, j],
Source_Matrix[i, j]);
                        // accumulate R = source_Matrix
                        for (h = j; h <= n; h++)
                        {
                            temp_R[1, -j + 1 + h] = cs[1] *
Source_Matrix[i - 1, h] - cs[2] * Source_Matrix[i, h];
                            temp_R[2, -j + 1 + h] = cs[2] *
Source_Matrix[i - 1, h] + cs[1] * Source_Matrix[i, h];
                        }
```

```
                        for (h = 1; h <= n - j + 1; h++)
                        {
                            Source_Matrix[i - 1, j + h - 1] = temp_R[1,
h];
                            Source_Matrix[i, j + h - 1] = temp_R[2, h];
                        }
                        // accumulate Q
                        for (h = 1; h <= m; h++)
                        {
                            temp_Q[h, 1] = cs[1] * Q[h, i - 1] - cs[2]
* Q[h, i];
                            temp_Q[h, 2] = cs[2] * Q[h, i - 1] + cs[1]
* Q[h, i];
                        }
                        for (h = 1; h <= m; h++)
                        {
                            Q[h, i - 1] = temp_Q[h, 1];
                            Q[h, i] = temp_Q[h, 2];
                        }
                    }
                }
            }
            if (null != done) done();
            return Q;
        }
```

Just as we encoded the Householder QR avoiding the explicit formation of Q, we can create an encoded Givens QR matrix where the zeroed elements have been replaced by a scalar that enables reconstruction of the Givens matrices (see Stewart). This scalar is calculated each time the Givens subroutine is called using the following encoder:

```
'cs is the returned result from the Givens function
If cs(1) = 0 Then 'This is the rho encoder section
   rho = 1
Else
   If Abs(cs(2)) <= Abs(cs(1)) Then 'here we want to include "="
      rho = Sign(cs(1)) * cs(2) / 2
   Else
      rho = 2 * Sign(cs(2)) / cs(1)
   End If
End If
```

Later, we can decode an entry in an encoded Givens QR and obtain the Givens matrix that was required to zero the element in any given position using this decoder:

```
If encoded_Givens_QR (i, j) = 1 Then
   cs(1) = 0 : cs(2) = 1
Else
   If Abs(encoded_Givens_QR (i, j)) < 1 Then
      cs(2) = 2 * encoded_Givens_QR (i, j) : cs(1) = (1 - (cs(2) * cs(2))) ^
     0.5
```

```
            Else
               cs(1) = 2 / encoded_Givens_QR (i, j) : cs(2) = (1 - (cs(1) * cs(1))) ^
              0.5
            End If
        End If
```

The following function returns an encoded Givens QR factorization:

```
        public double[,] Givens_QR_Simple_encoded(double[,]
Source_Matrix)
        {
            // 'Golub/Van Loan 5.2.2 Returns an encoded QR and
overwrites source_matrix with encoded R
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            QRRank = n; // by default
            int h = 0, i = 0, j = 0;
            double[] cs = new double[3];
            double rho = 0;
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(Source_Matrix[i, j]) >
Givens_Zero_Value)//  If it's already zero we won't zero it.
                    {
                        double[,] tempA = new double[3, n - j + 1 + 1];
                        cs = Givens(Source_Matrix[i - 1, j],
Source_Matrix[i, j]);
                        if (cs[1] == 0)
                        { // This is the rho encoder section
                            rho = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(cs[2]) <=
System.Math.Abs(cs[1]))
                            {
                                rho = System.Math.Sign(cs[1]) * cs[2] /
2;
                            }
                            else
                            {
                                rho = 2 * System.Math.Sign(cs[2]) /
cs[1];
                            }
                        }
                        // accumulate R
                        for (h = j; h <= n; h++)
                        {
                            tempA[1, -j + 1 + h] = cs[1] *
Source_Matrix[i - 1, h] - cs[2] * Source_Matrix[i, h];
                            tempA[2, -j + 1 + h] = cs[2] *
Source_Matrix[i - 1, h] + cs[1] * Source_Matrix[i, h];
                        }
```

```
                        for (h = 1; h <= n - j + 1; h++)
                        {
                            Source_Matrix[i - 1, j + h - 1] = tempA[1,
h];
                            Source_Matrix[i, j + h - 1] = tempA[2, h];
                        }
                        Source_Matrix[i, j] = rho;
                    }
                }
            }
            if (null != done) done();
            return Source_Matrix;
        }
```

An encoded Givens QR factorization can be decoded into Q and R using this function:

```
        public double[,] Decode_Givens_QR(double[,] Encoded_QR)
        {
            // Returns Q from encoded QR, R is just the upper triangle
of encoded_R.
            // Conserves Encoded_R
            int m = Encoded_QR.GetLength(0) - 1;
            int n = Encoded_QR.GetLength(1) - 1;
            double[,] local_encoded_QR = new double[m + 1, n + 1];
            Array.Copy(Encoded_QR, local_encoded_QR,
Encoded_QR.Length);
            int h = 0, i = 0, j = 0;
            double[] cs = new double[4];
            double[,] Q = new double[m + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                Q[i, i] = 1;
            }
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (local_encoded_QR[i, j] != 0)
                    { // Don't bother if rho is 0. It's just
multiplying by one.

                        // rho decoder section
                        if (local_encoded_QR[i, j] == 1)
                        {
                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(local_encoded_QR[i, j])
< 1)
                            {
                                cs[2] = 2 * local_encoded_QR[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
```

```
                            }
                            else
                            {
                                cs[1] = 2 / local_encoded_QR[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        // accumulate Q
                        double[,] temp_Q = new double[m + 1, 4];
                        for (h = 1; h <= m; h++)
                        {
                            temp_Q[h, 1] = cs[1] * Q[h, i - 1] - cs[2]
* Q[h, i];
                            temp_Q[h, 2] = cs[2] * Q[h, i - 1] + cs[1]
* Q[h, i];
                        }
                        for (h = 1; h <= m; h++)
                        {
                            Q[h, i - 1] = temp_Q[h, 1];
                            Q[h, i] = temp_Q[h, 2];
                        }
                    }
                }
            }
            return Q;
        }
```

The encoded Householder QR technique and the encoded Givens QR technique have similar accuracy characteristics. The Householder QR algorithm uses an arithmetic "butcher knife" to zero sub-diagonal elements, whereas the Givens algorithm uses a surgical approach to zero sub-diagonal elements. The Householder procedure is more efficient for a general QR, but the Givens procedure can be adapted to certain problems that have structural features where selective zeroing is either more efficient or simply required. For example, consider the case of a matrix with a sparsely populated lower triangle. It isn't necessary to execute the above subroutine's inner loop (For i = m To j + 1 Step -1) in its entirety if $R(i, j)$ — as we encounter it — is already zero.

Fast Givens QR Factorization

The fast Givens "QR" algorithm does not calculate Q and R directly. It calculates M, D, and T such that:

D is a diagonal matrix. Its elements are positive so we can take their square roots without getting into imaginary numbers.
T is upper triangular
$M^T A = T$ but $A \neq MT$
$M^T M = D$
$Q = MD^{-½}$ where $D^{-½}$ is a diagonal matrix with elements $1/\sqrt{d_{i,i}}$

$$R = D^{-1/2}T$$

In a standard QR factorization, Q is a new orthonormal basis for matrix A that enables re-expression of A in an upper triangular form R. In the MT factorization of A obtained using fast Givens, M is a basis but its vectors are neither orthogonal nor normal. But if Q and R are needed they can be determined.

The algorithm works in conjunction with an element zeroing function much like the Givens function. At any step in the zeroing process we seek the matrix $M_{ij} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}$ such that $M_{ij}^{T} \begin{pmatrix} x_{i-1,j} \\ x_{i,j} \end{pmatrix} = \begin{pmatrix} Update\ x_{i-1,j} \\ 0 \end{pmatrix}$. The M_{ij} takes on one of two alternate forms: $Type\ 1 = \begin{pmatrix} \beta & 1 \\ 1 & \alpha \end{pmatrix}$ or $Type\ 2 = \begin{pmatrix} 1 & \alpha \\ \beta & 1 \end{pmatrix}$. The zeroing function performs three tasks:

It returns α and β.
It returns the Type
It updates the $d_{i-1,j}$ and $d_{i,j}$ elements of D

An implementation of the algorithm for the fast Givens function is provided below:

```
public double[] FastGivens(double[] x, double[] d)
{
    // Golub/Van Loan 5.1.4
    // returns result(0)= alpha; Result(1)=Beta; result(2)=type
    double[] result = new double[4];
    double alpha = 0, beta = 0, gamma = 0, type = 0;
    if (x[2] != 0)
    {
        alpha = -x[1] / x[2];
        beta = -alpha * (d[2] / d[1]);
        gamma = -alpha * beta;
        if (gamma <= 1)
        {
            double tau = 0;
            type = 1;
            tau = d[1];
            d[1] = (1 + gamma) * d[2];
            d[2] = (1 + gamma) * tau;
        }
        else
        {
            type = 2;
            alpha = 1 / alpha;
            beta = 1 / beta;
            gamma = 1 / gamma;
            d[1] = (1 + gamma) * d[1];
            d[2] = (1 + gamma) * d[2];
        }
```

```
            }
            else
            {
                type = 2;
                alpha = 0;
                beta = 0;
            }
            result[0] = alpha;
            result[1] = beta;
            result[2] = type;
            return result;
        }
```

The fast Givens zeroing function has complexity similar to that of the Givens function except that it does not involve square roots. The fast Givens function is called by the fast Givens QR subroutine below to calculate M, D, and T.

```
        public void FastGivens_QR(double[,] Source_Matrix, ref
double[,] D_destination, ref double[,] M_destination, ref double[,]
T_destination)
        {
            // Golub/Van Loan 5.2.4 that explicitly forms M
            // Source_Matrix is overwritten by T. D, M, and T are
returned by reference
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            QRRank = n; // by default
            int i = 0, j = 0, k = 0;
            D_destination = new double[2, m + 1];
            M_destination = new double[m + 1, m + 1];
            // T_destination= T
            for (i = 1; i <= m; i++)
            {
                D_destination[1, i] = 1;
                M_destination[i, i] = 1;
            }
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(Source_Matrix[i, j]) >
Givens_Zero_Value)// No need to zero if already zero.
                    {
                        double alpha = 0;
                        double beta = 0;
                        int type = 0;
                        double[] fg_x = new double[4];
                        double[] fg_d = new double[4];
                        fg_x[1] = Source_Matrix[i - 1, j];
                        fg_x[2] = Source_Matrix[i, j];
                        fg_d[1] = D_destination[1, i - 1];
                        fg_d[2] = D_destination[1, i];
                        double[] Return_FG_result = new double[4];
                        Return_FG_result = FastGivens(fg_x, fg_d);
                        alpha = Return_FG_result[0];
                        beta = Return_FG_result[1];
```

```
                        type =
System.Convert.ToInt32(Return_FG_result[2]);
                        D_destination[1, i - 1] = fg_d[1];
                        D_destination[1, i] = fg_d[2];
                        double[,] tempA = new double[3, n - j + 1 + 1];
                        double[,] temp_bigm = new double[m + 1, 3];
                        if (type == 1)
                        {
                            for (k = j; k <= n; k++)
                            {
                                tempA[1, -j + 1 + k] = beta *
Source_Matrix[i - 1, k] + Source_Matrix[i, k];
                                tempA[2, -j + 1 + k] = Source_Matrix[i
- 1, k] + alpha * Source_Matrix[i, k];
                            }
                            for (k = 1; k <= m; k++)
                            {
                                temp_bigm[k, 1] = M_destination[k, i -
1] * beta + M_destination[k, i];
                                temp_bigm[k, 2] = M_destination[k, i -
1] + M_destination[k, i] * alpha;
                            }
                        }
                        else
                        {
                            for (k = j; k <= n; k++)
                            {
                                tempA[1, -j + 1 + k] = Source_Matrix[i
- 1, k] + beta * Source_Matrix[i, k];
                                tempA[2, -j + 1 + k] = alpha *
Source_Matrix[i - 1, k] + Source_Matrix[i, k];
                            }
                            for (k = 1; k <= m; k++)
                            {
                                temp_bigm[k, 1] = M_destination[k, i -
1] + M_destination[k, i] * beta;
                                temp_bigm[k, 2] = M_destination[k, i -
1] * alpha + M_destination[k, i];
                            }

                        }
                        for (k = 1; k <= n - j + 1; k++)
                        {
                            Source_Matrix[i - 1, j + k - 1] = tempA[1,
k];
                            Source_Matrix[i, j + k - 1] = tempA[2, k];
                        }
                        for (k = 1; k <= m; k++)
                        {
                            M_destination[k, i - 1] = temp_bigm[k, 1];
                            M_destination[k, i] = temp_bigm[k, 2];
                        }
                    }
                }
            }
            T_destination = Source_Matrix;
            if (null != done) done();
```

```
}
```

In the above procedure, Source_Matrix is transformed to T through a sequence of zeroing transformations involving multiplication by M_{ij}^{T} .Similarly, M is explicitly accumulated by performing the transformations in sequence on an appropriately dimensioned identity matrix. Either of these operations involves two multiplications and two additions for each step. The standard Givens QR requires four multiplications and two additions. So, the explicit fast Givens QR is faster than the explicit Givens QR.

The fast Givens procedure has certain overflow/underflow hazards associated with it. Anda and Park provide guidance for dealing with this problem.

QR Factorization by the Modified Gram-Schmidt Method

If we have $A \in \mathbb{R}^{mxn}$ where $m \geq n$, its QR factorization results in $Q \in \mathbb{R}^{mxm}$ and $R \in \mathbb{R}^{mxn}$. In many cases we only need the first n columns of Q. We define a *thin* QR factorization of A as $Q(1 \text{ to } m,\ 1 \text{ to } n)R(1 \text{ to } n,\ 1 \text{ to } n)$. The modified Gram-Schmidt method (MGS) produces a thin QR factorization of A. I will not go into any further detail about the method. Here is an MGS implementation:

```
        public void MGS(double[,] Source_Matrix, ref double[,] Q1, ref
double[,] R1)
        {
            // Golub/Van Loan 5.2.5
            // destroys Source_matrix, returns Q1 and R1
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            int i = 0, j = 0, k = 0;
            Q1 = new double[m + 1, n + 1];
            R1 = new double[n + 1, n + 1];
            double norm = 0;
            for (k = 1; k <= n; k++)
            {
                norm = 0;
                for (i = 1; i <= m; i++)
                {
                    norm = norm + (Math.Pow(Source_Matrix[i, k], 2));
                }
                norm = Math.Pow(norm, 0.5);
                R1[k, k] = norm;
                for (i = 1; i <= m; i++)
                {
                    Q1[i, k] = Source_Matrix[i, k] / R1[k, k];
                }
                for (j = k + 1; j <= n; j++)
                {
                    double[,] qt = new double[n + 1, m + 1];
                    qt = Transpose(Q1);
                    for (i = 1; i <= m; i++)
                    {
```

```
                        R1[k, j] = R1[k, j] + qt[k, i] *
Source_Matrix[i, j];
                    }
                    for (i = 1; i <= m; i++)
                    {
                        Source_Matrix[i, j] = Source_Matrix[i, j] -
Q1[i, k] * R1[k, j];
                    }
                }
            }
            QRRank = n - 1; //  by default -1 because its Source_Matrix
should be ab augmented
            if (null != done) done();
        }
```

Solving Full Rank, Over Determined Systems

Here we consider the system $Ax = b$ where $A \in \mathbb{R}^{m,n}$, x is an n dimensional vector, b is an m dimensional vector, and $m > n$. There will not necessarily be any vector x that is an exact solution for all the equations in the system. Our task is to find an x that best fits the data. We will focus on the *least squares* best fit.

Typically we have a data matrix A and an observation vector b, and we want to determine a coefficient vector x. For example, consider the following experimental data (or process data):

Stimulus Level, Z	Response, b
1.00	2.014339
1.50	2.281294
2.00	3.03292
3.00	6.027431
5.00	18.18885
7.00	38.07983

Let's say we know enough about this system to be able to conjecture that the response is a quadratic function of the stimulus, i.e.,

$\text{Constant } Z^0 + \text{Coefficient}_1 Z^1 + \text{Coefficient}_2 Z^2 = b$.

We want to know x_1=Constant, x_2=Coefficient$_1$ and x_3=Coefficient$_2$. Notice that in this case the first element is a constant. We first construct our data matrix. The first column will be a column of ones (Z^0) — the multiplier for the constant. The next column will be our stimulus values for each of the six test points (Z^1). The last column will be the stimulus values for each of the six test points squared (Z^2).This general type of matrix where we have columns (or rows) consisting of increasing powers of Z is called a Vandermonde matrix. We have:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 6 & 36 \end{pmatrix} \qquad b = \begin{pmatrix} -2 \\ -9 \\ -22 \\ -41 \\ -66 \\ -97 \end{pmatrix}$$

The whole problem in matrix format reduces to:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 6 & 36 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -2 \\ -9 \\ -22 \\ -41 \\ -66 \\ -97 \end{pmatrix},$$

representing the system of simultaneous equations:

$$\begin{aligned} x_1 + x_2 + x_3 &= -2 \\ x_1 + 2x_2 + 4x_3 &= -9 \\ x_1 + 3x_2 + 9x_3 &= -22 \\ x_1 + 4x_2 + 16x_3 &= -41 \\ x_1 + 5x_2 + 25x_3 &= -66 \\ x_1 + 6x_2 + 36x_3 &= -97 \end{aligned}$$

Now for any vector $x = \left(x_1 + x_2 + x_3\right)^T$ we can calculate a *residual* vector using the formula $Ax - b$. Remember that the Euclidian norm of a vector is the sum of the squares of the elements. If we determine x such that the norm $\|Ax - b\|_2$ is minimized, then that x is the best *least squares fit* for our data.

It need not be the case that the A matrix is a Vandermonde or any other structured matrix. Column one could be ambient temperature, column two could be relative humidity, and column three could be pounds of corn starch added. The observation vector could be viscosity of the fudge batch at a candy factory.

So, how do we find the vector x that minimizes the residual norm? We solve $Ax = b$, giving each row in A and b (each set of simultaneous equations) equal consideration.

The Method of Normal Equations

Recall that for the determined system $Ax = b$ we were able to multiply both sides of the equation by A^{-1}, giving $x = A^{-1}b$. We cannot do that with an over determined system because A cannot be inverted (it is not square). But $A^T A$ is square and can be inverted. We will multiply both sides by A^T, giving $A^T Ax = A^T b$. Now we will multiply both sides by $\left(A^T A\right)^{-1}$, giving $x = (A^T A)^{-1} A^T b$. That's all there is to it! Compare this matrix computation development with the tedious development given for normal equations techniques in most statistics texts. The matrix approach is one of simple beauty.

A procedure based on the ideas discussed above, but which avoids an explicit inverse through a Cholesky factorization ($A^T A$ is symmetric positive definite), is encapsulated below.

```
        public double[,] Normal_Ax_b_by_Cholesky_onestep(double[,]
A_Matrix, double[,] b)
        {
            // Golub/Van Loan 238 Solves system using normal equations
by Cholesky method
            int i = 0, j = 0;
            int m = A_Matrix.GetLength(0) - 1;
            int n = A_Matrix.GetLength(1) - 1;
            // compute lower triangle of ATA
            double[,] At = null;
            At = Transpose(A_Matrix);
            double[,] c = null;
            c = Matrix_Multiply(At, A_Matrix);
            for (i = 1; i <= n; i++)
            {
                for (j = i + 1; j <= n; j++)
                {
                    c[i, j] = 0;
                }
            }
            // compute Result=ATb
            double[,] Result = null;
            Result = Matrix_Multiply(At, b);
            // Cholesky factor C
            double[,] G = null;
            G = Cholesky(c);
            // solve Gy=x(Result) forward sub
            double Temp = 0;
            Result[1, 1] = Result[1, 1] / G[1, 1];
            for (i = 2; i <= n; i++)
            {
                for (j = 1; j <= i - 1; j++)
                {
                    Temp = Temp - G[i, j] * Result[j, 1];
```

```
            }
            Temp = Temp + Result[i, 1];
            Result[i, 1] = Temp / G[i, i];
            Temp = 0;
        }
        // solve GTx=y backward sub
        Temp = 0;
        double[,] Gtrans = null;
        Gtrans = Transpose(G);
        Temp = 0;
        Result[n, 1] = Result[n, 1] / Gtrans[n, n];
        for (i = n - 1; i >= 1; i += -1)
        {
            for (j = n; j >= i + 1; j += -1)
            {
                Temp = Temp - Gtrans[i, j] * Result[j, 1];
            }
            Temp = Temp + Result[i, 1];
            Result[i, 1] = Temp / Gtrans[i, i];
            Temp = 0;
        }
        if (null != done) done();
        return Result;
    }
```

As written, this method does not produce a reusable factor. The algorithm from which it is derived requires $(m+n/3)n^2$ FLOPs.

There is an important shortcoming in the normal equations method. Any time we explicitly form the product of a matrix and its transpose, we risk a severe loss of information (see Watkins, exercise 3.5.25). A more consistently accurate technique involves the QR factorization.

Householder and Givens QR Solutions

In solving $Ax = b$ we know that $QR = A$ so $QRx = b$ and $Rx = Q^T b$. The expression $Q^T b$ is a vector of length m: $Q^T b = \left[\frac{c}{d}\right]\begin{matrix}\text{1 to n}\\ \text{n+1 to m}\end{matrix}$. Likewise, R is an upper trapezoidal mxn matrix: $R = \left[\frac{R_{n,n}}{0}\right]\begin{matrix}\text{1 to n}\\ \text{n+1 to m}\end{matrix}$. The solution can be obtained by using back calculation on the following system: $(R_{nn})(x) = (c)$. This process is encapsulated in the following function:

```
public double[,] QR_Ax_b(double[,] Q, double[,] R, double[,] b)
{
    // Golub/Van Loan pg 259 and 239 Given Q, R, b solve for x
    int i = 0, j = 0, k = 0;
    int m = Q.GetLength(0) - 1;
    int n = R.GetLength(1) - 1;
```

```
            // Make QTb. A thin Q is used here.
            double[,] Qtb = new double[n + 1, 2];
            for (i = 1; i <= n; i++)
            {
                for (k = 1; k <= m; k++)
                {
                    Qtb[i, 1] = Q[k, i] * b[k, 1] + Qtb[i, 1]; // note
here that Q(k,i)=QT(i,k)
                }
            }
            // QTAPz=QTb and QTAP is Upper triangular so by back
calculation
            double[,] z = new double[R.GetLength(1) - 1 + 1, 2];
            z[n, 1] = Qtb[n, 1] / R[n, n]; // note that the first n X n
rows and cols of R = QTAP
            for (i = n - 1; i >= 1; i += -1)
            {
                for (j = n; j >= i + 1; j += -1)
                {
                    z[i, 1] = z[i, 1] - R[i, j] * z[j, 1];
                }
                z[i, 1] = z[i, 1] + Qtb[i, 1];
                z[i, 1] = z[i, 1] / R[i, i];
            }
            if (null != done) done();
            return z;
        }
```

Note in the above function that although we have a complete mxm Q, we only use the first n columns (a thin Q) to form $Q^T b$.

Is explicit formation of the Q matrix necessary for solving $Ax = QRx = b$? Can we somehow use our encoded matrix directly to obtain a solution? Yes we can, and our approach will exact a computational effort savings beyond that gained by eliminating the explicit formation of Q and R. The Householder vectors stored in the lower triangle of our encoded matrix represent a factored form of Q that can be used directly to obtain $Q^T b$ in a matrix–vector — rather than a matrix–matrix — multiplication. A similar argument can be made for the Givens QR procedure. There are two functions below. One takes an encoded Householder QR and b as arguments, and another takes an encoded Givens QR and b as arguments.

Householder:

```
        public double[,] Householder_Encoded_QR_Ax_b(double[,]
Encoded_QR, double[,] b)
        {
            // Golub/Van Loan pp 259 and 239 Given encoded QR, b solve
for x. Does not overwrite QR or b.
            int i = 0, j = 0;
            int m = Encoded_QR.GetLength(0) - 1;
            int n = Encoded_QR.GetLength(1) - 1;
            double[,] QTb = new double[m + 1, 2];
            Array.Copy(b, QTb, b.Length);
```

```
            for (j = 1; j <= n; j++)
            { // Hb=(I-BetavvT)b=A-v(BetabTv)T Golub/Van Loan p 211

                double[,] v = new double[m + 1, 2];
                v[j, 1] = 1;
                for (i = j + 1; i <= m; i++)
                {
                    v[i, 1] = Encoded_QR[i, j];
                }
                double[,] omega = new double[2, 2];
                omega = Matrix_Multiply(Transpose(QTb), v);
                for (i = 1; i <= m; i++)
                {
                    QTb[i, 1] = QTb[i, 1] - 1 * beta[j] * omega[1, 1] *
v[i, 1]; // This beta(j) is a global
                }
            }
            // QTAPx=QTb and QTAP is the Upper triangular nXn
supermatrix of R (aka r11), So by back calculation
            double[,] z = new double[Encoded_QR.GetLength(1) - 1 + 1,
2];
            z[n, 1] = QTb[n, 1] / Encoded_QR[n, n];
            for (i = n - 1; i >= 1; i += -1)
            {
                for (j = n; j >= i + 1; j += -1)
                {
                    z[i, 1] = z[i, 1] - Encoded_QR[i, j] * z[j, 1];
                }
                z[i, 1] = z[i, 1] + QTb[i, 1];
                z[i, 1] = z[i, 1] / Encoded_QR[i, i];
            }
            // z holds  x
            if (null != done) done();
            return z;
        }
```

Givens:

```
        public double[,] Givens_Encoded_QR_Ax_b(double[,] Encoded_QR,
double[,] b)
        {
            int m = Encoded_QR.GetLength(0) - 1;
            int n = Encoded_QR.GetLength(1) - 1;
            int i = 0, j = 0;
            double[,] x = new double[n + 1, 2];
            double[] cs = new double[4];
            double[,] QTb = new double[m + 1, 2];
            Array.Copy(b, QTb, b.Length);
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (Encoded_QR[i, j] != 0)// Don't bother if rho is
0. It's just multiplying by one.
                    {
                        if (Encoded_QR[i, j] == 1)
```

```
                        { //  this is the rho decoder section
                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(Encoded_QR[i, j]) < 1)
                            {
                                cs[2] = 2 * Encoded_QR[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
                            else
                            {
                                cs[1] = 2 / Encoded_QR[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute QTb
                        double temp2 = 0;
                        temp1 = cs[1] * QTb[i - 1, 1] - cs[2] * QTb[i,
1];
                        temp2 = cs[2] * QTb[i - 1, 1] + cs[1] * QTb[i,
1];
                        QTb[i - 1, 1] = temp1;
                        QTb[i, 1] = temp2;
                    }
                }
            }
            x[n, 1] = QTb[n, 1] / Encoded_QR[n, n]; // solve for x by
back calculation
            for (i = n - 1; i >= 1; i += -1)
            {
                for (j = n; j >= i + 1; j += -1)
                {
                    x[i, 1] = x[i, 1] - Encoded_QR[i, j] * x[j, 1];
                }
                x[i, 1] = x[i, 1] + QTb[i, 1];
                x[i, 1] = x[i, 1] / Encoded_QR[i, i];
            }
            if (null != done) done();
            return x;
        }
```

Fast Givens Solutions

The fast Givens QR method does not give us Q and R. It gives us M, T and D, from which we can calculate Q and R. However, it is not necessary to form Q and R to solve $Ax = b$. We can use M and T directly. As detailed in Golub/Van Loan:

$$M^T A = \begin{bmatrix} S \\ 0 \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix} \text{ and } MTb = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix} \text{ so } Sx = c.$$

The following function puts these relationships to work for us:

```
        public double[,] FastGivins_Ax_b(double[,] b, double[,] _M,
double[,] T)
        {
            // Golub/Van Loan p 241
            int m = T.GetLength(0) - 1;
            int n = T.GetLength(1) - 1;
            int i = 0, j = 0;
            double[,] result = new double[n + 1, 2];
            double[,] MT = new double[m + 1, m + 1];
            MT = Transpose(_M);
            double[,] cd = new double[m + 1, 2];
            cd = Matrix_Multiply(MT, b);
            result[n, 1] = cd[n, 1] / T[n, n];
            double temp = 0;
            for (i = n - 1; i >= 1; i += -1)
            {
                for (j = n; j >= i + 1; j += -1)
                {
                    temp = temp - T[i, j] * result[j, 1];
                }
                temp = temp + cd[i, 1];
                result[i, 1] = temp / T[i, i];
                temp = 0;
            }
            if (null != done) done();
            return result;
        }
```

Stewart's note, in addition to giving a method for economical storage of Givens rotators, also briefly mentions application of the method to fast Givens procedures. The implementation problem, however, seems to be a refractory problem. My search of the literature and discussions with colleagues indicate no such implementation is in general use. There are certainly ways to store the three critical variables (alpha, beta and type) for each step in the zeroing process in separate arrays. Below I have provided a fast Givens solver that takes the A matrix and b vector and efficiently (it forms $M^T b$ implicitly) solves for x. It is not reusable in the sense that multiple solutions can be obtained with a single pass through the factorization logic.

```
        public double[,] FastGivens_QR_onestep(double[,] A, double[,]
b)
        {
            // Implicitly solves Ax=b by applyling transformations to b
as they are developed.
            // A is overwritten by T.
            int m = A.GetLength(0) - 1;
            QRRank = A.GetLength(1) - 1;//by default
            int n = A.GetLength(1) - 1;
            int i = 0, j = 0, k = 0;
            double[,] QTb = new double[m + 1, 2];
            double[,] Result = new double[n + 1, 2];
            Array.Copy(b, QTb, b.Length);
            double[] d = new double[m + 1];
            for (i = 1; i <= m; i++)
```

```
            {
                d[i] = 1;
            }
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(A[i, j]) > Givens_Zero_Value)
                    { // No need to zero if already zero.

                        double alpha = 0;
                        double beta = 0;
                        int type = 0;
                        double[] fg_x = new double[4];
                        double[] fg_d = new double[4];
                        fg_x[1] = A[i - 1, j];
                        fg_x[2] = A[i, j];
                        fg_d[1] = d[i - 1];
                        fg_d[2] = d[i];
                        double[] Return_FG_result = new double[4];
                        Return_FG_result = FastGivens(fg_x, fg_d);
                        alpha = Return_FG_result[0];
                        beta = Return_FG_result[1];
                        type =
System.Convert.ToInt32(Return_FG_result[2]);
                        d[i - 1] = fg_d[1];
                        d[i] = fg_d[2];
                        double[,] tempA = new double[4, n - j + 1 + 1];
                        double temp1 = 0;
                        double temp2 = 0;
                        if (type == 1)
                        {
                            for (k = j; k <= n; k++)
                            {
                                tempA[1, -j + 1 + k] = beta * A[i - 1,
k] + A[i, k];

                                tempA[2, -j + 1 + k] = A[i - 1, k] +

alpha * A[i, k];
                            }
                            // compute Q transpose b
                            temp1 = QTb[i - 1, 1] * beta + QTb[i, 1];
                            temp2 = QTb[i - 1, 1] + QTb[i, 1] * alpha;
                            QTb[i - 1, 1] = temp1;
                            QTb[i, 1] = temp2;
                        }
                        else
                        {
                            for (k = j; k <= n; k++)
                            {
                                tempA[1, -j + 1 + k] = A[i - 1, k] +
beta * A[i, k];

                                tempA[2, -j + 1 + k] = alpha * A[i - 1,

k] + A[i, k];
                            }
                            // compute Q transpose b
                            temp1 = QTb[i - 1, 1] + QTb[i, 1] * beta;
                            temp2 = QTb[i - 1, 1] * alpha + QTb[i, 1];
```

```
                        QTb[i - 1, 1] = temp1;
                        QTb[i, 1] = temp2;
                    }
                    for (k = 1; k <= n - j + 1; k++)
                    {
                        A[i - 1, j + k - 1] = tempA[1, k];
                        A[i, j + k - 1] = tempA[2, k];
                    }
                }
            }
        }
        Result[n, 1] = QTb[n, 1] / A[n, n]; // solve for result by
back calculation
        for (i = n - 1; i >= 1; i += -1)
        {
            for (j = n; j >= i + 1; j += -1)
            {
                Result[i, 1] = Result[i, 1] - A[i, j] * Result[j,
1];
            }
            Result[i, 1] = Result[i, 1] + QTb[i, 1];
            Result[i, 1] = Result[i, 1] / A[i, i];
        }
        if (null != done) done();
        return Result;
    }
```

Again, users beware. Fast Givens procedures have overflow/underflow risks.

Modified Gram-Schmidt Solutions

As explained earlier, the MGS method returns a thin QR factorization, $A = Q_1 R_1$. An algorithm that uses this factorization directly to obtain a solution for x in Ax=b is unstable. A more accurate algorithm results from forming an augmented matrix $[Ab]$ and submitting this matrix to the MGS decomposition function given above. As detailed in Golub/Van Loan, the result has the form:

$$A_{augmented} = [Ab] = [Q_1 q_{n+1}] \begin{bmatrix} R1 & z \\ 0 & \rho \end{bmatrix}, \; z = Q_1^T b \text{ and } R_1 x = z$$

To implement a solution we need a function to make augmented matrices:

```
    public double[,] MakeAugmentedArray(double[,] MatrixA,
double[,] MatrixB)
    {
        int i = 0, j = 0;
        int mA = MatrixA.GetLength(0) - 1;
        int nA = MatrixA.GetLength(1) - 1;
        int mB = MatrixB.GetLength(0) - 1;
        int nB = MatrixB.GetLength(1) - 1;
```

```
            if (mA != mB)
            {
                throw (new ApplicationException("A & B must have equal
number of rows"));
            }
            double[,] result = new double[mA + 1, nA + nB + 1];
            for (i = 1; i <= mA; i++)
            {
                for (j = 1; j <= nA; j++)
                {
                    result[i, j] = MatrixA[i, j];
                }
            }
            for (i = 1; i <= mA; i++)
            {
                for (j = nA + 1; j <= nA + nB; j++)
                {
                    result[i, j] = MatrixB[i, j - nA];
                }
            }
            if (null != done) done();
            return result;
        }
```

Then we need to submit that augmented matrix to the MGS function and by back calculation solve for x. We will need a function for back calculation:

```
        public double[,] Back_calculate_x(double[,]
Upper_Triangular_Matrix, double[,] b)
        {
            int n = Upper_Triangular_Matrix.GetLength(1) - 1;
            double[,] local_b = new double[n + 1, 2];
            Array.Copy(b, local_b, b.Length);
            double temp = 0;
            int i = 0, j = 0;
            local_b[n, 1] = local_b[n, 1] / Upper_Triangular_Matrix[n,
n];
            for (i = n - 1; i >= 1; i += -1)
            {
                for (j = n; j >= i + 1; j += -1)
                {
                    temp = temp - Upper_Triangular_Matrix[i, j] *
local_b[j, 1];
                }
                temp = temp + b[i, 1];
                local_b[i, 1] = temp / Upper_Triangular_Matrix[i, i];
                temp = 0;
            }
            if (null != done) done();
            return local_b;
        }
```

For demonstration purposes, the entire implementation will be orchestrated from an interface class in the final section of this chapter. Before we leave MGS solutions, it should be noted that the technique has no advantages over Householder and Givens QR

solutions. It is more computationally expensive and, because we are performing the factorization on the augmented matrix $[Ab]$, it does not provide a reusable factor.

An Example

Consider the well conditioned system Ax=b where:

$$A = \begin{bmatrix} -0.72 & 0.78 & -0.93 & 0.42 \\ 0.6 & -0.42 & -0.55 & -0.72 \\ 0.4 & -0.33 & -0.63 & 0.13 \\ 0.49 & -0.78 & 0.48 & 0.83 \\ -0.49 & -0.96 & 0.86 & 0.55 \\ 0.13 & 0.69 & 0.81 & 0.53 \end{bmatrix}, \text{ and } b = \begin{bmatrix} 0.1425 \\ -5.2155 \\ -1.425 \\ 4.294 \\ 2.774 \\ 6.2605 \end{bmatrix}$$

The Householder QR Methods yield:

$$QR_{encoded} = \begin{bmatrix} 1.239153 & -0.6195361 & 0.005406919 & -0.3843755 \\ -0.3062548 & 1.586182 & -0.5826885 & -0.2904675 \\ -0.2041699 & 0.02805388 & 1.688444 & 0.6260355 \\ -0.2501081 & 0.2725491 & -0.1259701 & 1.166175 \\ 0.2501081 & 0.8304145 & -0.5213985 & -2.619453 \\ -0.06635521 & -0.496249 & -0.2485669 & -3.258441 \end{bmatrix}$$

$$Q = \begin{bmatrix} -0.5810421 & 0.2648015 & -0.4575582 & 0.4802244 & -0.2972368 & 0.2528748 \\ 0.4842017 & -0.07566565 & -0.3534067 & -0.2869364 & -0.6821278 & 0.2954595 \\ 0.3228011 & -0.08196599 & -0.402445 & 0.4135 & -0.04950719 & -0.7441007 \\ 0.3954314 & -0.3372979 & 0.1666163 & 0.6686068 & 0.158273 & 0.4796017 \\ -0.3954314 & -0.7596755 & 0.248444 & 0.0187022 & -0.4128687 & -0.18437 \\ 0.1049104 & 0.4759829 & 0.643659 & 0.2620783 & -0.4983875 & -0.1762441 \end{bmatrix}$$

$$R = \begin{bmatrix} 1.239153 & -0.6195361 & 0.005406919 & -0.3843755 \\ 0 & 1.586182 & -0.5826885 & -0.2904675 \\ 0 & 0 & 1.688444 & 0.6260355 \\ 0 & 0 & 0 & 1.166175 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$x = \{0.95 \quad 1.9 \quad 2.85 \quad 4.75\}^T$$

The Givens QR Methods yield:

$$QR_{encoded} = \begin{bmatrix} -1.239153 & 0.6195361 & -0.005406919 & 0.3843755 \\ 3.442091 & 1.586182 & -0.5826885 & -0.2904675 \\ 3.361712 & 6.147047 & -1.688444 & -0.6260355 \\ -4.053085 & 8.07967 & 3.363904 & 1.166175 \\ 2.877768 & -7.062371 & -10.38621 & 0.3153294 \\ 0.1282174 & 0.1509458 & 4.473432 & 0.1419598 \end{bmatrix}$$

$$Q = \begin{bmatrix} 0.5810421 & 0.2648015 & 0.4575582 & 0.4802244 & 0.3902504 & 0 \\ -0.4842017 & -0.07566565 & 0.3534067 & -0.2869364 & 0.7109992 & 0.2169671 \\ -0.3228011 & -0.08196599 & 0.402445 & 0.4135 & -0.4444556 & 0.5988289 \\ -0.3954314 & -0.3372979 & -0.1666163 & 0.6686068 & 0.1902231 & -0.4678497 \\ 0.3954314 & -0.7596755 & -0.248444 & 0.0187022 & 0.1949959 & 0.4079577 \\ -0.1049104 & 0.4759829 & -0.643659 & 0.2620783 & 0.2653973 & 0.4571831 \end{bmatrix}$$

$$R = \begin{bmatrix} -1.239153 & 0.6195361 & -0.005406919 & 0.3843755 \\ 0 & 1.586182 & -0.5826885 & -0.2904675 \\ 0 & 0 & -1.688444 & -0.6260355 \\ 0 & 0 & 0 & 1.166175 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$x = \{0.95 \quad 1.9 \quad 2.85 \quad 4.75\}^T$$

The fast Givens method yields:

$$M = \begin{bmatrix} 1.469388 & -0.4396498 & -0.7035707 & 1 & 0.6862774 & 0 \\ -1.22449 & 0.1256277 & -0.5434206 & -0.5975049 & 1.250332 & 0.3032593 \\ -0.8163266 & 0.1360881 & -0.6188251 & 0.8610557 & -0.7816004 & 0.8369952 \\ -1 & 0.5600156 & 0.2561999 & 1.39228 & 0.3345182 & -0.653923 \\ 1 & 1.261289 & 0.3820234 & 0.03894471 & 0.3429114 & 0.5702108 \\ -0.2653061 & -0.7902743 & 0.9897311 & 0.5457414 & 0.4667162 & 0.6390141 \end{bmatrix}$$

$$T = \begin{bmatrix} -3.133673 & 1.566735 & -0.01367347 & 0.9720408 \\ 0 & -2.633538 & 0.9674375 & 0.4822631 \\ 0 & 0 & 2.59626 & 0.9626322 \\ 0 & 0 & 0 & 2.428396 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 6.395252 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.756596 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.36441 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4.336223 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3.092522 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.953622 \end{bmatrix}$$

$$x = \begin{Bmatrix} 0.95 & 1.9 & 2.85 & 4.75 \end{Bmatrix}^T$$

Also, $Q = MD^{1/2}\mathrm{Q}$ and $R = D^{1/2}T$ are the same matrices as obtained with the standard Givens procedure.

The Matrix Computation Utility Class

I mentioned in Chapter 1 that we are developing a class library for matrix computations, and provided a list of global variables for that class. In Chapter 2 and in this chapter, we developed methods (functions and subprocedures) for the class library. Now that we have something to work with, let us take a moment to see how we can interface with this class from another *user interface* program. I have named the matrix computation utility class Matrix_Computation_Utility. Here are the global declarations for the main program:

```
internal Matrix_Comp_Library_C.Matrix_Computation_Utility solver = new
Matrix_Comp_Library_C.Matrix_Computation_Utility();
public double[,] A;
public double[,] B;
public double[,] C;
public double[,] D;
public double[,] E;
public double[,] F;
public double[,] G;
public double[,] XY;
public double[,] x;
```

This demonstration will involve implementing our algorithm for solving for x in Ax=b using the MGS approach discussed above. First we need to form $Ab_{augmented}$ and factor it:

```
private void factor_MGS_Click(object sender, EventArgs e)
{
  C = solver.MakeAugmentedArray(A, B);
  solver.MGS(C, ref D, ref E); //Q1=Q1qnplus1 and R1=R1z
 }
```

Now we will solve for x:

```
private void solve_MGS_Click (object sender, EventArgs e)
{
    int n = E.GetLength(1) - 2;
    int i = 0, j = 0;
    double[,] R1 = new double[n + 1, n + 1];
    double[,] z = new double[n + 1, 2];
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)
        {
            R1[i, j] = E[i, j];
        }
    }
    for (i = 1; i <= n; i++)
    {
        z[i, 1] = E[i, n + 1];
    }
    x = solver.Back_calculate_x(R1, z);
 }
```

Let's see the results from our previous example using this MGS approach:

$$[Ab] = \begin{bmatrix} -0.72 & 0.78 & -0.93 & 0.42 & 0.1425 \\ 0.6 & -0.42 & -0.55 & -0.72 & -5.2155 \\ 0.4 & -0.33 & -0.63 & 0.13 & -1.425 \\ 0.49 & -0.78 & 0.48 & 0.83 & 4.294 \\ -0.49 & -0.96 & 0.86 & 0.55 & 2.774 \\ 0.13 & 0.69 & 0.81 & 0.53 & 6.2605 \end{bmatrix}$$

$$Q_1 q_{n+1} = \begin{bmatrix} -0.5810421 & 0.2648015 & -0.4575582 & 0.4802244 & 0.2326673 \\ 0.4842017 & -0.07566565 & -0.3534067 & -0.2869364 & 0.5816682 \\ 0.3228011 & -0.08196599 & -0.402445 & 0.4135 & 0 \\ 0.3954314 & -0.3372979 & 0.1666163 & 0.6686068 & 0 \\ -0.3954314 & -0.7596755 & 0.248444 & 0.0187022 & 0.6252933 \\ 0.1049104 & 0.4759829 & 0.643659 & 0.2620783 & 0.4653346 \end{bmatrix}$$

$$R_1 z = \begin{bmatrix} 1.239153 & -0.6195361 & 0.005406919 & -0.3843755 & -1.810297 \\ 0 & 1.586182 & -0.5826885 & -0.2904675 & -0.02663594 \\ 0 & 0 & 1.688444 & 0.6260355 & 7.785736 \\ 0 & 0 & 0 & 1.166175 & 5.53933 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$x = \begin{Bmatrix} 0.95 & 1.9 & 2.85 & 4.75 \end{Bmatrix}^T$$

Some Timing Experiments

As mentioned in the introduction, some of our procedures are not entirely optimized. For example, look at the Householder QR methods and the House function and ask, "How can the process of submitting the vectors to House, obtaining a Householder vector, and — in the case of the encoded method — packing the return into the sub diagonal elements of our return matrix be simplified for efficiency and compressed with respect to memory requirements?" Clearly there are many opportunities. With a little index switching we could put the House function inline within the QR method and process the submatrix columns directly, updating the original elements of the source matrix where appropriate. This would not be as intuitive and easy to follow, but it would save the few moments of processing time being wasted converting the submatrix columns to vectors, processing those vectors, then populating a new vector with leading zeros with those reprocessed vectors. Also, consider the fact that we represent vectors as matrices and we treat matrix vector multiplications as matrix–matrix multiplications. There are many ways to further streamline our code. Although we have sacrificed some efficiency, we have been careful not to commit serious blunders like forming Householder matrices and failing to truncate Givens matrix operations.

Note in the various implementations provided so far that we have raised an event when the procedure completes: `if (null != done) done();`. This event enables timing experiemnts to be conducted. We first set a global variable `timedouble = DateTime.Now.ToOADate();` when the method is called. Then we handle the done event with:

```
public double done_eventhandler()
{
   double X = 24 * 60 * 60 * (DateTime.Now.ToOADate() - timedouble);
   return X;
}
```

To evaluate timing we need test matrices; different procedures will perform with higher relative efficiency depending on the test matrix structure. In specific applications, the procedure used should consider these differences.

Our first test matrix is a thoroughly populated (no zero elements) 200 × 200 element symmetric positive definite matrix. This will give us a chance to relate timings

for determined systems using the LU and Cholesky methods from Chapter 2 to timings using our over determined systems methods. The general behavior we will observe for the over determined methods is typical of nearly square systems. We will determine total time to reach a result for a system Ax=b, where b in this case is a 200 element vector. The b vector was pre-calculated from a known x vector, so we know that the norm $\|Ax-b\|_2$ has a theoretical value of zero. The 2-norm of b is 163.

Method	Time in milliseconds	$\|Ax - b\|_2$
LU	46	3.08E-09
Cholesky	15	2.81E-09
Normal Equations	124	3.67E-04
Fast Givens Oneshot	180	6.47E-09
Householder QR explicit	1250	6.73E-10
Householder QR implicit	690	4.38E-10
Givens QR explicit	468	7.63E-09
Givens QR implicit	281	9.38E-09
MGS – Augmented system	25000	3.43E-09

Ideally, the Householder procedures should outperform the Givens procedures, but evidently implementation overhead is masking the underlying higher efficiency of the Householder methods. The MGS method is clearly not the method of choice for nearly square systems. If there are no accuracy sacrifices, our Chapter 2 methods should be used for determined systems. Finally, in this example, the normal equations method is showing its instability as machine precision is ~1E -15.

Now let's consider a tall thin system. Our test matrix will be a thoroughly populated 50,000 × 3 matrix. Again $\|Ax-b\|_2$ has a theoretical value of zero. The 2-norm of b is 67600. We won't test the explicit QR techniques.

Method	Time in milliseconds	$\|Ax - b\|_2$
Normal Equations	16	3.43E-09
Fast Givens Oneshot	250	3.43E-09
Householder QR implicit	125	3.43E-09
Givens QR implicit	250	3.43E-09
MGS – Augmented system	109	3.43E-09

Here we see the Householder technique's ability to zero entire columns at once begin to have a pronounced effect. The MGS method is more competitive with the Householder technique, because the length of the column of elements being worked on by the two procedures is essentially the same. Again, the 2X advantage of the fast Givens procedure is being lost in the implementation.

Finally, if we take the matrix above and zero all but the first three rows of the first column, and all but the first four rows of the second column, we will obtain a sparse matrix.

Method	Time in milliseconds
Householder QR implicit	126
Givens QR implicit	78

Our Givens procedure, which ignores elements that are already zero, now takes the lead.

The matrices we have been looking at are relatively small. Although today's microprocessor clock speeds are stellar, memory limitations are severe. Clearly, avoiding the formation of Q by using our implicit procedures is quite important. With smaller matrices, procedural infrastructure (dimensioning arrays, calling functions, throwing exceptions, etc.) constitutes a fair percentage of the problem. As matrices become larger and larger, the differences in speed among the different procedures becomes more and more pronounced. At some point we would expect to reap the benefits of the fast Givens advantage, but for typical problems that are going to be adaptable to memory limited systems, the primary choices for linear least squares solutions are the Givens and Householder QR procedures.

4. QR Factorizations with Pivoting, Complete Orthogonalization and Rank Deficient, Over Determined Systems

Rank Deficiency

Consider the following matrix: $A = \begin{bmatrix} 1 & 3 & 4 \\ 2 & 5 & 7 \\ 3 & 7 & 10 \\ 4 & 9 & 13 \end{bmatrix}$.

The third column is not linearly independent of one and two (it is the sum of one and two). The $Rank(A)$ is the number of linearly independent columns or rows of A. To deal with cases where the $Rank(A)$ may be less than n we need a factorization procedure that discovers and appropriately deals with this *rank deficiency*.

Factorization Methods

Householder QR with Column Pivoting

One solution is a Householder QR decomposition with column pivoting. The procedure begins with determining the Euclidian norm for each column, then exchanges columns to move the column with the highest norm into the first position. Then the Householder matrix and the intermediary Q and R matrices are determined as usual. Next we re-evaluate the norms for the submatrix below the first row and to the right of the first column. Again, the columns are exchanged (the whole column). The value of the highest norm is stored in a variable, tau. Tau is compared with a machine error multiple that we can set to some fixed value. If tau is greater than the threshold then the process will repeat, otherwise it will terminate. (When an orthogonal transformation is performed on two dependent vectors their rows will zero in a similar fashion, giving rise to a zero norm calculation (in exact arithmetic) for the second of the two when it is encountered.) The number of iterations for the process is stored in an index called QRRank. All of the column swapping is recorded in a permutation matrix.

Recall that for a simple QR factorization $Q^T A = R$, when we perform a QR with column pivoting as detailed above, we obtain a factorization with the following structure:

$$Q^T AP = R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$$

1 to Rank Rank+1 to n

where P is the column permutation matrix.

The algorithm accomplishes three things: it evaluates the rank, it moves dependent column vectors to the righthand side of the matrix (beyond column = rank), and it records the exchanges in a permutation matrix. Otherwise, the output is the same as usual. Perhaps in theory we should set the threshold value τ to zero, but with floating point arithmetic we need to use some multiple of machine precision. In the code below we use the general rule of thumb that the appropriate threshold is $\|A\|_F \varepsilon$. This rule does have exceptions, and solutions by way of the singular value decomposition may be better. Here is an encapsulation that uses as a threshold value the product of the matrix F norm and the machine precision value:

```
        public double[,] Householder_QR_Pivot(double[,] Source_Matrix)
        {
            // Explicit column pivot QR adapted from Golub/Van Loan
5.4.1
            // Overwrites source_matrix with R and returns Q from
function call. Modifies QR_Encoded_P
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            beta = new double[n + 1];
            int i = 0, j = 0, k = 0;
            double[] c = new double[n + 1];
            double[,] Q = new double[m + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                Q[i, i] = 1;
            }
            QR_Encoded_P = new int[n + 1];
            for (i = 1; i <= n; i++)
            {
                QR_Encoded_P[i] = i;
            }
            for (j = 1; j <= n; j++)
            { // Determine Euclidian norm for each column.
                for (i = 1; i <= m; i++)
                {
                    c[j] = c[j] + (Math.Pow(Source_Matrix[i, j], 2));
                }
            }
            double fnorm = 0;
            for (i = 1; i <= n; i++)
            {
                fnorm = fnorm + c[i];
            }
            fnorm = Math.Pow(fnorm, 0.5);
            double tau = 0;
            // find column with highest norm and enter that norm in tau
            QRRank = 0;
            tau = Machine_Error * fnorm;
            for (i = 1; i <= n; i++)
            {
                if ((c[i]) > tau)
                {
                    tau = c[i];
                    k = i;
```

```
                }
            }
            while (tau > Machine_Error * fnorm)
            {
                QRRank = QRRank + 1;
                // column swap
                if (k != QRRank)
                {
                    double[] ColTemp = new double[m + 1];
                    double cTemp = 0;
                    cTemp = c[k];
                    c[k] = c[QRRank];
                    c[QRRank] = cTemp;
                    for (i = 1; i <= m; i++)
                    {// for the Source_Matrix matrix
                        ColTemp[i] = Source_Matrix[i, QRRank];
                        Source_Matrix[i, QRRank] = Source_Matrix[i, k];
                        Source_Matrix[i, k] = ColTemp[i];
                    }
                    int temp = 0;
                    temp = QR_Encoded_P[k]; // for the permutation
matrix
                    QR_Encoded_P[k] = QR_Encoded_P[QRRank];
                    QR_Encoded_P[QRRank] = temp;
                }
                // Convert the rankth column to a 1 dimensional vector
for submission to house.
                double[] vector = new double[m - QRRank + 1 + 1];
                double[,] v = new double[m + 1, 2];
                for (i = 1; i <= vector.Length - 1; i++)
                {
                    vector[i] = Source_Matrix[QRRank + i - 1, QRRank];
                }
                // Submit it
                vector = House(vector);
                // Convert the result to a column vector with leading
zeros up to the QRRank row.
                for (i = 1; i <= vector.Length - 1; i++)
                {
                    v[QRRank + i - 1, 1] = vector[i];
                }
                beta[QRRank] = HOUSE_BETA;
                double[,] omegaT = null;
                omegaT = Matrix_Multiply(v,
Scalar_multiply(Transpose(Matrix_Multiply(Transpose(Source_Matrix),
v)), beta[QRRank]));
                for (i = QRRank; i <= m; i++)
                {
                    for (k = QRRank; k <= n; k++)
                    {
                        Source_Matrix[i, k] = Source_Matrix[i, k] -
omegaT[i, k];
                    }
                }
                double[,] omega = null;
```

```
                omega =
Scalar_multiply(Matrix_Multiply(Matrix_Multiply(Q, v), Transpose(v)),
beta[QRRank]);
                for (i = 1; i <= m; i++)
                {
                    for (k = 1; k <= m; k++)
                    {
                        Q[i, k] = Q[i, k] - omega[i, k];
                    }
                }
                for (i = QRRank + 1; i <= n; i++)
                { // Golub/Van Loan p. 249

                    c[i] = c[i] - (Source_Matrix[QRRank, i] *
Source_Matrix[QRRank, i]);
                }
                if (QRRank < n)
                {
                    tau = Machine_Error * fnorm;
                    for (i = QRRank + 1; i <= n; i++)
                    {
                        if (c[i] > tau)
                        {
                            tau = c[i];
                            k = i;
                        }
                    }
                }
                else
                {
                    tau = 0;
                }
            }
            if (null != done) done();
            return Q;
        }
```

Just as we did for the QR simple algorithm, we can modify the column pivot QR procedure to give an encoded QR factorization where the upper triangle is R and the elements below the principal diagonal are the essential elements of the Householder vectors. We will also have to return the permutation matrix.

```
        public double[,] Householder_QR_Pivot_encoded(double[,]
Source_Matrix)
        {
            // Encoded column pivot QR adapted from Golub/Van Loan
5.4.1 Form encoded QR without explicit formation of Householder
matrices
            // Overwrites source_matrix with encoded QR and also
returns encoded QR. Modifies QR_Encoded_P.
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            int i = 0, j = 0, k = 0;
            double[] c = new double[n + 1];
            QR_Encoded_P = new int[n + 1];
```

```
            for (i = 1; i <= n; i++)
            {
                QR_Encoded_P[i] = i;
            }
            beta = new double[n + 1];
            for (j = 1; j <= n; j++)
            { // Determine Euclidian norm for each column.
                for (i = 1; i <= m; i++)
                {
                    c[j] = c[j] + (Math.Pow(Source_Matrix[i, j], 2));
                }
            }
            double fnorm = 0;
            for (i = 1; i <= n; i++)
            {
                fnorm = fnorm + c[i];
            }
            fnorm = Math.Pow(fnorm, 0.5);
            double tau = 0;
            // Find column with highest norm and enter that norm in
tau.
            QRRank = 0;
            tau = Machine_Error * fnorm;
            for (i = 1; i <= n; i++)
            {
                if (c[i] > tau)
                {
                    tau = c[i];
                    k = i;
                }
            }
            while (tau > Machine_Error * fnorm)
            {
                QRRank = QRRank + 1;
                if (k != QRRank)
                { // swap columns
                    double[] ColTemp = new double[m + 1];
                    double cTemp = 0;
                    cTemp = c[k];
                    c[k] = c[QRRank];
                    c[QRRank] = cTemp;
                    for (i = 1; i <= m; i++)
                    {// for the Source_Matrix matrix
                        ColTemp[i] = Source_Matrix[i, QRRank];
                        Source_Matrix[i, QRRank] = Source_Matrix[i, k];
                        Source_Matrix[i, k] = ColTemp[i];
                    }
                    int temp2 = 0;
                    temp2 = QR_Encoded_P[k]; // for the permutation
matrix
                    QR_Encoded_P[k] = QR_Encoded_P[QRRank];
                    QR_Encoded_P[QRRank] = temp2;
                }
                // Convert the rankth column to a 1 dimensional vector
for submission to house.
                double[] vector = new double[m - QRRank + 1 + 1];
                double[,] v = new double[m + 1, 2];
```

```
                for (i = 1; i <= vector.Length - 1; i++)
                {
                    vector[i] = Source_Matrix[QRRank + i - 1, QRRank];
                }
                // Submit it
                vector = House(vector);
                // Convert the result to a column vector with leading
zeros up to the QRRank row.
                for (i = 1; i <= vector.Length - 1; i++)
                {
                    v[QRRank + i - 1, 1] = vector[i];
                }
                beta[QRRank] = HOUSE_BETA;
                double[,] omegaT = null;
                omegaT = Matrix_Multiply(v,
Scalar_multiply(Transpose(Matrix_Multiply(Transpose(Source_Matrix),
v)), beta[QRRank]));
                for (i = QRRank; i <= m; i++)
                {
                    for (k = QRRank; k <= n; k++)
                    {
                        Source_Matrix[i, k] = Source_Matrix[i, k] -
omegaT[i, k];
                    }
                }
                // Place the essential part of the Householder vectors
below the principal diagonal
                for (i = QRRank + 1; i <= m; i++)
                {
                    Source_Matrix[i, QRRank] = vector[i - QRRank + 1];
                }
                for (i = QRRank + 1; i <= n; i++)
                { // Golub/Van Loan p. 249
                    c[i] = c[i] - (Source_Matrix[QRRank, i] *
Source_Matrix[QRRank, i]);
                }
                if (QRRank < n)
                {
                    tau = Machine_Error * fnorm;
                    for (i = QRRank + 1; i <= n; i++)
                    {
                        if (c[i] > tau)
                        {
                            tau = c[i];
                            k = i;
                        }
                    }
                }
                else
                {
                    tau = 0;
                }
            }
            if (null != done) done();
            return Source_Matrix;
        }
```

Givens QR with Column Pivoting

Inspection of the Householder procedures shows a simple Householder QR embedded in a rank negotiating framework. We can use any other QR technique such as Givens or fast Givens. Here is the explicit Givens QR with column pivoting procedure:

```
        public double[,] Givens_QR_Pivot(double[,] Source_Matrix)
        {
            // Explicit column pivot QR: Overwrites source_matrix with
R and returns Q from function call. Modifies QR_Encoded_P
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            int h = 0, i = 0, j = 0, k = 0;
            double[] c = new double[n + 1];
            double[,] Q = new double[m + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                Q[i, i] = 1;
            }
            double[] cs = new double[4];
            QR_Encoded_P = new int[n + 1];
            for (i = 1; i <= n; i++)
            {
                QR_Encoded_P[i] = i;
            }
            for (j = 1; j <= n; j++)
            { // Determine Euclidian norm for each column.
                for (i = 1; i <= m; i++)
                {
                    c[j] = c[j] + (Math.Pow(Source_Matrix[i, j], 2));
                }
            }
            double fnorm = 0;
            for (i = 1; i <= n; i++)
            {
                fnorm = fnorm + c[i];
            }
            fnorm = Math.Pow(fnorm, 0.5);
            double tau = 0;
            // find column with highest norm and enter that norm in tau
            QRRank = 0;
            tau = Machine_Error * fnorm;
            for (i = 1; i <= n; i++)
            {
                if ((c[i]) > tau)
                {
                    tau = c[i];
                    k = i;
                }
            }
            while (tau > Machine_Error * fnorm)
            {
                QRRank = QRRank + 1;
                // column swap
                if (k != QRRank)
```

```
                {
                    double[] ColTemp = new double[m + 1];
                    double cTemp = 0;
                    cTemp = c[k];
                    c[k] = c[QRRank];
                    c[QRRank] = cTemp;
                    for (i = 1; i <= m; i++)
                    {
                        ColTemp[i] = Source_Matrix[i, QRRank];
                        Source_Matrix[i, QRRank] = Source_Matrix[i, k];
                        Source_Matrix[i, k] = ColTemp[i];
                    }
                    int temp = 0;
                    temp = QR_Encoded_P[k]; // for the permutation
matrix
                    QR_Encoded_P[k] = QR_Encoded_P[QRRank];
                    QR_Encoded_P[QRRank] = temp;
                }
                j = QRRank;
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(Source_Matrix[i, j]) >
Givens_Zero_Value)
                    { //  If it's already zero we won't zero it.
                        double[,] tempA = new double[3, n - j + 1 + 1];
                        double[,] temp_Q = new double[m + 1, 3];
                        cs = Givens(Source_Matrix[i - 1, j],
Source_Matrix[i, j]);
                        // accumulate Source_Matrix
                        for (h = j; h <= n; h++)
                        {
                            tempA[1, -j + 1 + h] = cs[1] *
Source_Matrix[i - 1, h] - cs[2] * Source_Matrix[i, h];
                            tempA[2, -j + 1 + h] = cs[2] *
Source_Matrix[i - 1, h] + cs[1] * Source_Matrix[i, h];
                        }
                        for (h = 1; h <= n - j + 1; h++)
                        {
                            Source_Matrix[i - 1, j + h - 1] = tempA[1,
h];
                            Source_Matrix[i, j + h - 1] = tempA[2, h];
                        }
                        // accumulate Q
                        for (h = 1; h <= m; h++)
                        {
                            temp_Q[h, 1] = cs[1] * Q[h, i - 1] - cs[2]
* Q[h, i];
                            temp_Q[h, 2] = cs[2] * Q[h, i - 1] + cs[1]
* Q[h, i];
                        }
                        for (h = 1; h <= m; h++)
                        {
                            Q[h, i - 1] = temp_Q[h, 1];
                            Q[h, i] = temp_Q[h, 2];
                        }
                    }
                }
```

```
                for (i = QRRank + 1; i <= n; i++)
                { // Golub/Van Loan p. 249

                    c[i] = c[i] - (Source_Matrix[QRRank, i] *
Source_Matrix[QRRank, i]);
                }
                if (QRRank < n)
                {
                    tau = Machine_Error * fnorm;
                    for (i = QRRank + 1; i <= n; i++)
                    {
                        if (c[i] > tau)
                        {
                            tau = c[i];
                            k = i;
                        }
                    }
                }
                else
                {
                    tau = 0;
                }
            }
            if (null != done) done();
            return Q;
        }
```

The corresponding encoded Givens QR with column pivoting is as follows:

```
        public double[,] Givens_QR_Pivot_encoded(double[,]
Source_Matrix)
        {
            // Explicit column pivot Givens QR.
            // Overwrites source_matrix with encoded QR and also
returns encoded QR. Modifies QR_Encoded_P.
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            int h = 0, i = 0, j = 0, k = 0;
            double rho = 0;
            double[] c = new double[n + 1];
            double[] cs = new double[3];
            QR_Encoded_P = new int[n + 1];
            for (i = 1; i <= n; i++)
            {
                QR_Encoded_P[i] = i;
            }
            for (j = 1; j <= n; j++)
            { // Determine Euclidian norm for each column.
                for (i = 1; i <= m; i++)
                {
                    c[j] = c[j] + (Math.Pow(Source_Matrix[i, j], 2));
                }
            }
            double fnorm = 0;
            for (i = 1; i <= n; i++)
            {
```

```
                fnorm = fnorm + c[i];
            }
            fnorm = Math.Pow(fnorm, 0.5);
            double tau = 0;
            // find column with highest norm and enter that norm in tau
            QRRank = 0;
            tau = Machine_Error * fnorm;
            for (i = 1; i <= n; i++)
            {
                if ((c[i]) > tau)
                {
                    tau = c[i];
                    k = i;
                }
            }
            while (tau > Machine_Error * fnorm)
            {
                QRRank = QRRank + 1;
                // column swap
                if (k != QRRank)
                {
                    double[] ColTemp = new double[m + 1];
                    double cTemp = 0;
                    cTemp = c[k];
                    c[k] = c[QRRank];
                    c[QRRank] = cTemp;
                    for (i = 1; i <= m; i++)
                    {
                        ColTemp[i] = Source_Matrix[i, QRRank];
                        Source_Matrix[i, QRRank] = Source_Matrix[i, k];
                        Source_Matrix[i, k] = ColTemp[i];
                    }
                    int temp = 0;
                    temp = QR_Encoded_P[k]; // for the permutation
matrix
                    QR_Encoded_P[k] = QR_Encoded_P[QRRank];
                    QR_Encoded_P[QRRank] = temp;
                }
                j = QRRank;
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(Source_Matrix[i, j]) >
Givens_Zero_Value)
                    { //  If it's already zero we won't zero it.

                        double[,] tempA = new double[3, n - j + 1 + 1];
                        double[,] temp_Q = new double[m + 1, 3];
                        cs = Givens(Source_Matrix[i - 1, j],
Source_Matrix[i, j]);
                        if (cs[1] == 0)
                        { // This is the rho encoder section
                            rho = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(cs[2]) <=
System.Math.Abs(cs[1]))
```

```
                            {
                                rho = System.Math.Sign(cs[1]) * cs[2] /
2;
                            }
                            else
                            {
                                rho = 2 * System.Math.Sign(cs[2]) /
cs[1];
                            }
                        }
                        // accumulate R
                        for (h = j; h <= n; h++)
                        {
                            tempA[1, -j + 1 + h] = cs[1] *
Source_Matrix[i - 1, h] - cs[2] * Source_Matrix[i, h];
                            tempA[2, -j + 1 + h] = cs[2] *
Source_Matrix[i - 1, h] + cs[1] * Source_Matrix[i, h];
                        }
                        for (h = 1; h <= n - j + 1; h++)
                        {
                            Source_Matrix[i - 1, j + h - 1] = tempA[1,
h];
                            Source_Matrix[i, j + h - 1] = tempA[2, h];
                        }
                        Source_Matrix[i, j] = rho; //  put encoded Q
below diagonal
                    }
                }
                for (i = QRRank + 1; i <= n; i++)
                { // Golub/Van Loan p. 249
                    c[i] = c[i] - (Source_Matrix[QRRank, i] *
Source_Matrix[QRRank, i]);
                }
                if (QRRank < n)
                {
                    tau = Machine_Error * fnorm;
                    for (i = QRRank + 1; i <= n; i++)
                    {
                        if (c[i] > tau)
                        {
                            tau = c[i];
                            k = i;
                        }
                    }
                }
                else
                {
                    tau = 0;
                }
            }
            if (null != done) done();
            return Source_Matrix;
        }
```

The Complete Orthogonal Decomposition

The QR factorization with column pivoting gives a factorization of form:

$$Q^T AP = R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$$

1 to Rank Rank+1 to n

As detailed in Golub/Van Loan, the above form can be further reduced to zero R_{12} by forming $\begin{bmatrix} R^T_{11} \\ R^T_{12} \end{bmatrix}$ and performing a simple QR factorization on it. We will denote Q for this second factorization as Z. As expected, this results in an upper triangular matrix $Z^T \begin{bmatrix} R^T_{11} \\ R^T_{12} \end{bmatrix} = \begin{bmatrix} T^T_{11} \\ 0 \end{bmatrix}$. We may now combine this result with the original factorization to obtain:

$$Q^T APZ = \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$$

1 to Rank Rank+1 to n

So, obtaining a complete orthogonalization involves a QR with pivoting, followed by a simple QR. Either Householder or Givens techniques can be used. If we use the Householder procedure we will want to store the beta values for both decompositions. An advantage of the Givens approach is that we do not need to maintain these two additional vectors. Implementations of both approaches are provided below.

```
        public void Complete_orthogonal_implicit_Householder(double[,]
A_matrix, ref double[,] QR_destination, ref double[,] TZ, ref double[,]
QRBeta, ref double[,] TZBeta)
        {
            // Golub/Van Loan p 250. Overwrites A_matrix with encoded
QR.
            int m = A_matrix.GetLength(0) - 1;
            int n = A_matrix.GetLength(1) - 1;
            int rank = 0;
            int i = 0, j = 0;
            QR_destination = Householder_QR_Pivot_encoded(A_matrix);
            rank = QRRank;
            QRBeta = new double[rank + 1, 2]; //  Array for QR beta
            for (j = 1; j <= rank; j++)
            {
                QRBeta[j, 1] = beta[j];
            }
            TZ = new double[n + 1, rank + 1]; //  array for T encoded
            // TZ(n, rank) n X rank transpose of encoded QR.
Transposing allows submission to our QR Simple Encoded Function
            for (i = 1; i <= rank; i++)
```

```
            {
                for (j = i; j <= n; j++)
                {
                    TZ[j, i] = A_matrix[i, j];
                }
            }
            TZ = Householder_QR_Simple_encoded(TZ);
            TZ = Transpose(TZ); // TZ now holds the left to right QR
the upper rank X rank square of which is T11
            TZBeta = new double[rank + 1, 2]; // Array for T beta
            for (j = 1; j <= rank; j++)
            {
                TZBeta[j, 1] = beta[j];
            }
            QRRank = rank; // reset to original QR rank
            if (null != done) done();
        }


        public void Complete_orthogonal_implicit_Givens(double[,]
A_matrix, ref double[,] QR_destination, ref double[,] ZT)
        {
            // Golub/Van Loan p 250. Overwrites A_matrix with encoded
QR.
            int m = A_matrix.GetLength(0) - 1;
            int n = A_matrix.GetLength(1) - 1;
            int rank = 0;
            int i = 0, j = 0;
            QR_destination = Givens_QR_Pivot_encoded(A_matrix);
            rank = QRRank;
            ZT = new double[n + 1, rank + 1];
            // T(n, rank) n X rank transpose of encoded QR. Transposing
allows submission to our QR Simple Encoded Function
            for (i = 1; i <= rank; i++)
            {
                for (j = i; j <= n; j++)
                {
                    ZT[j, i] = A_matrix[i, j];
                }
            }
            ZT = Givens_QR_Simple_encoded(ZT); // t now holds QR
            QRRank = rank; // reset to original QR rank
            if (null != done) done();
        }
```

Be aware that in all of the procedures above the permutation matrix, QR_Encoded_P(1), and QRRank are being modified and will be needed later on.

Obtaining Solutions With Pivoting QR Factorizations

Rank deficient least squares problems have an infinite number of least squares solutions. The QR with column pivoting algorithms give us one of them. The solutions

they give have at most Rank(A) number of non-zero elements. Again, the QR factorization with column pivoting gives a factorization of form:

$$Q^T AP = R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$$
$$\text{1 to Rank} \quad \text{Rank+1 to n}$$

In a least squares solution, because $Rx = Q^T b$, we have:

$$\begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix} \begin{bmatrix} x \end{bmatrix} = Q^T b = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$$
$$\text{1 to Rank} \quad \text{Rank+1 to n}$$

If we restrict our attention to the upper left Rank × Rank block of R (i.e., R_{11}) and the first Rank rows of $Q^T b$ (i.e., c), our basic solution will reduce to $R_{11} x = c$ (where x is a Rank dimensional vector), which we can solve by back calculation. We then append x with zeros to form an n-dimensional vector $\begin{pmatrix} x \\ 0 \end{pmatrix}$. The only thing left to do is multiply the result by the permutation matrix. So, in a sense, we ignore all but one member of any set of dependent columns in A. In a set of dependent columns in our coefficient matrix A, the only one for which the corresponding x multiplier is determined not to be zero is the one with the initial highest norm.

For explicit QR with column pivoting solutions we have:

```
        public double[,] QR_pivot_Ax_b(double[,] Q, double[,] R,
double[,] b)
        {
            // Golub/Van Loan pg 259 and 239 Given Q, R, b solve for x
            int i = 0, j = 0, k = 0;
            int m = Q.GetLength(0) - 1;
            int n = R.GetLength(1) - 1;
            int Rank = QRRank;
            // make QTb. A thin Q is used here.
            double[,] Qtb = new double[Rank + 1, 2]; // QTb=c
            for (i = 1; i <= Rank; i++)
            {
                for (k = 1; k <= m; k++)
                {
                    Qtb[i, 1] = Q[k, i] * b[k, 1] + Qtb[i, 1]; // note
here that Q(k,i)=QT(i,k)
                }
            }
            // QTAPz=QTb and QTAP is Upper triangular so by back
calculation
            double[,] z = new double[R.GetLength(1) - 1 + 1, 2];
            z[Rank, 1] = Qtb[Rank, 1] / R[Rank, Rank]; // note that the
first rank X rank rows and cols of R = QTAP=r11
```

```
            for (i = Rank - 1; i >= 1; i += -1)
            {
                for (j = Rank; j >= i + 1; j += -1)
                {
                    z[i, 1] = z[i, 1] - R[i, j] * z[j, 1];
                }
                z[i, 1] = z[i, 1] + Qtb[i, 1];
                z[i, 1] = z[i, 1] / R[i, i];
            }
            // Multiply by permutation matrix but do not use multiply
function
            double[,] tempb = new double[n + 1, 2]; //  multiply by
permutation
            for (i = 1; i <= n; i++)
            {
                tempb[QR_Encoded_P[i], 1] = z[i, 1];
            }
            if (null != done) done();
            return tempb;
        }
```

For solutions using an encoded Householder QR with column pivoting we have:

```
        public double[,] Householder_Encoded_QR_Pivot_Ax_b(double[,]
Encoded_QR, double[,] b)
        {
            // Golub/Van Loan pg 259 and 239 Given encoded QR, b [and
QR encoded Permutation from global] solve for x
            int i = 0, j = 0;
            int m = Encoded_QR.GetLength(0) - 1;
            int n = Encoded_QR.GetLength(1) - 1;
            int rank = QRRank;
            double[,] Qtb = new double[m + 1, 2];
            Array.Copy(b, Qtb, b.Length);
            double[,] Identity = new double[m + 1, m + 1];
            for (j = 1; j <= rank; j++)
            { // Hb=(I-BetavvT)b=A-v(BetabTv)T Golub/Van Loan p 211
                double[,] v = new double[m + 1, 2];
                v[j, 1] = 1;
                for (i = j + 1; i <= m; i++)
                {
                    v[i, 1] = Encoded_QR[i, j];
                }
                double[,] omega = new double[2, 2];
                omega = Matrix_Multiply(Transpose(Qtb), v);
                for (i = 1; i <= m; i++)
                {
                    Qtb[i, 1] = Qtb[i, 1] - 1 * beta[j] * omega[1, 1] *
v[i, 1]; // This beta(j) is a global
                }
            }
            // QTAPx=QTb and QTAP is the Upper triangular rankXrank
supermatrix of R aka r11, So by back calculation
            double[,] z = new double[Encoded_QR.GetLength(1) - 1 + 1,
2];
            z[rank, 1] = Qtb[rank, 1] / Encoded_QR[rank, rank];
```

```
            for (i = rank - 1; i >= 1; i += -1)
            {
                for (j = rank; j >= i + 1; j += -1)
                {
                    z[i, 1] = z[i, 1] - Encoded_QR[i, j] * z[j, 1];
                }
                z[i, 1] = z[i, 1] + Qtb[i, 1];
                z[i, 1] = z[i, 1] / Encoded_QR[i, i];
            }
            // z holds unpermuted x
            // Multiply by permutation matrix but do not use multiply
function
            double[,] tempb = new double[n + 1, 2]; //  multiply by
permutation
            for (i = 1; i <= n; i++)
            {
                tempb[QR_Encoded_P[i], 1] = z[i, 1];
            }
            if (null != done) done();
            return tempb;
        }
```

Finally, for solutions using an encoded Givens QR with column pivoting we have:

```
        public double[,] Givens_Encoded_QR_Pivot_Ax_b(double[,]
Encoded_QR, double[,] b)
        {
            int m = Encoded_QR.GetLength(0) - 1;
            int n = Encoded_QR.GetLength(1) - 1;
            int rank = QRRank;
            int i = 0, j = 0;
            double[,] x = new double[n + 1, 2];
            double[] cs = new double[4];
            double[,] QTb = new double[m + 1, 2];
            Array.Copy(b, QTb, b.Length);
            for (j = 1; j <= rank; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (Encoded_QR[i, j] != 0)// Don't bother if rho is
0. It's just multiplying by one.
                    {
                        if (Encoded_QR[i, j] == 1)
                        { //  this is the rho decoder section
                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(Encoded_QR[i, j]) < 1)
                            {
                                cs[2] = 2 * Encoded_QR[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
                            else
```

```
                            {
                                cs[1] = 2 / Encoded_QR[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute QTb=c
                        double temp2 = 0;
                        temp1 = cs[1] * QTb[i - 1, 1] - cs[2] * QTb[i,
1];
                        temp2 = cs[2] * QTb[i - 1, 1] + cs[1] * QTb[i,
1];
                        QTb[i - 1, 1] = temp1;
                        QTb[i, 1] = temp2;
                    }
                }
            }
            double[,] z = new double[Encoded_QR.GetLength(1) - 1 + 1,
2];
            z[rank, 1] = QTb[rank, 1] / Encoded_QR[rank, rank];
            for (i = rank - 1; i >= 1; i += -1)
            {
                for (j = rank; j >= i + 1; j += -1)
                {
                    z[i, 1] = z[i, 1] - Encoded_QR[i, j] * z[j, 1];
                }
                z[i, 1] = z[i, 1] + QTb[i, 1];
                z[i, 1] = z[i, 1] / Encoded_QR[i, i];
            }
            // z holds unpermuted x
            // Multiply by permutation matrix but do not use multiply
function
            double[,] tempb = new double[n + 1, 2]; //  multiply by
permutation
            for (i = 1; i <= n; i++)
            {
                tempb[QR_Encoded_P[i], 1] = z[i, 1];
            }
            if (null != done) done();
            return tempb;
        }
```

An example:

Consider the system Ax=b where:

$$A=\begin{bmatrix}1 & 1.5 & 1 & 2\\ 2 & 3 & 3 & 4\\ 3 & 4.5 & 2 & 6\\ 4 & 6 & 5 & 8\\ 5 & 7.5 & 4 & 10\end{bmatrix} \text{ and } b=\{12,27,33,51,57\}^T$$

Note that columns 1, 2 and 4 are dependent (column 2 = 1.5 × column 1 and column 4 = 2 × column 1). This is a rank 2 system and column 4 has the highest norm of the three dependent columns.

The Householder QR with column pivoting yields:

$$QR_{enc} = \begin{bmatrix} 14.8324 & 7.146519 & 11.1243 & 7.416198 \\ -0.311711 & 1.981735 & 0 & 0 \\ -0.4675666 & 0.973525 & 0 & 0 \\ -0.6234221 & -1.301281 & 0 & 0 \\ -0.7792776 & 0.8798804 & 0 & 0 \end{bmatrix}, \quad P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$Q = \begin{bmatrix} 0.13484 & 0.0183494 & 0.1598433 & 0.8664116 & 0.453059 \\ 0.2696799 & 0.5413072 & -0.4908056 & 0.3193753 & -0.5397856 \\ 0.4045199 & -0.4495602 & 0.4959569 & 0.1687347 & -0.5998455 \\ 0.5393599 & 0.578006 & 0.4741899 & -0.3071744 & 0.2361944 \\ 0.6741999 & -0.4128614 & -0.5125725 & -0.1565338 & 0.2962542 \end{bmatrix}$$

$$R = \begin{bmatrix} 14.8324 & 7.146519 & 11.1243 & 7.416198 \\ 0 & 1.981735 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \text{ and } x = \{0, 0, 3, 4.5\}^T$$

The Givens QR with column pivoting yields:

$$QR_{enc} = \begin{bmatrix} 14.8324 & 7.146519 & 11.1243 & 7.416198 \\ 14.8324 & -1.981735 & 0 & 0 \\ -7.348469 & -108 & 0 & 0 \\ 4.714045 & 3.550217 & 0 & 0 \\ -3.201562 & 3.901133 & 0 & 0 \end{bmatrix}, \quad P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$Q = \begin{bmatrix} 0.13484 & -0.0183494 & 0.9906974 & 0 & 0 \\ 0.2696799 & -0.5413072 & -0.04673101 & 0.7950317 & 0 \\ 0.4045199 & 0.4495602 & -0.04673101 & 0.166126 & 0.04746458 \\ 0.5393599 & -0.578006 & -0.08411583 & -0.5814411 & 0.1727737 \\ 0.6741999 & 0.4128614 & -0.08411583 & 0.04746458 & -0.6047079 \end{bmatrix}$$

$$R = \begin{bmatrix} 14.8324 & 7.146519 & 11.1243 & 7.416198 \\ 0 & -1.981735 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \text{ and } x = \{0, 0, 3, 4.5\}^T$$

Again, of the three dependent columns of A, only the column with the highest norm — column 4 — has a non zero multiplier in x.

Using Complete Orthogonalization to Obtain Solutions

The result obtained with the QR pivoting techniques may not be the best result for certain applications. The best result may be a least squares solution that is also a minimum Euclidian norm solution. In our QR column pivot solution we have selectively ignored the R_{12} term. In complete orthogonalization we further process the R matrix to zero R_{12}, and our solutions are minimum Euclidian norm solutions (see Golub and Van Loan, p. 259, for details).

As detailed in Golub/Van Loan, recall that our complete orthogonal factorization methods — both the Householder and Givens versions — produced a factored form solution for Q and T in the following framework:

$$Q^T APZ = T = \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$$
$$\text{1 to Rank} \quad \text{Rank+1 to n}$$

The form of our problem is not AZx=b, it is Ax=b. We can remedy this because ZZ^T is the identity matrix for an orthogonal Z. We have:

$$Q^T APZZ^T x = TZ^T x = \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix} Z^T \begin{bmatrix} x \end{bmatrix} = Q^T b$$
$$\text{1 to Rank} \quad \text{Rank+1 to n}$$

Now, $Z^T x$ has the form $Z^T x = \begin{bmatrix} w \\ y \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$ and $Q^T b$ has the form $Q^T b = \begin{bmatrix} c \\ d \end{bmatrix} \begin{matrix} \text{1 to Rank} \\ \text{Rank+1 to m} \end{matrix}$. Over the first rank elements of our system our equation becomes $T_{11} w = c$, so $x_{LS} = Z \begin{bmatrix} T_{11}^{-1} c \\ 0 \end{bmatrix}$, where c is a vector of the first rank elements of $Q^T b$.

In practice we first form c, the first rank elements of $Q^T b$. Next we solve the system $T_{11}w=c$ for w, where T_{11} is lower triangular using forward substitution. Using this approach we avoid explicitly forming T_{11}^{-1}. Next we apply Z to w to obtain the pre-permuted x_{ls}. Finally, we apply the permutation matrix.

Here are the solution functions for the Householder and Givens variants of the complete orthogonalization.

```
        public double[,]
Solve_Householder_complete_orthogonal_implicit(double[,] QR_encoded,
double[,] T_encoded, double[,] QR_Beta, double[,] T_beta, double[,] b)
        {
            // Golub/Van Loan p 250.
            int m = QR_encoded.GetLength(0) - 1;
            int n = QR_encoded.GetLength(1) - 1;
            int rank = T_encoded.GetLength(0) - 1;
            double[,] QTb = new double[m + 1, 2];
            Array.Copy(b, QTb, b.Length);
            int i = 0, j = 0;
            for (j = 1; j <= rank; j++)
            { // form QTb implicitly

                double[,] v = new double[m + 1, 2];
                v[j, 1] = 1;
                for (i = j + 1; i <= m; i++)
                {
                    v[i, 1] = QR_encoded[i, j];
                }
                double[,] omega = new double[2, 2];
                omega = Matrix_Multiply(Transpose(QTb), v);
                for (i = 1; i <= m; i++)
                {
                    QTb[i, 1] = QTb[i, 1] - 1 * QR_Beta[j, 1] *
omega[1, 1] * v[i, 1]; // This beta(j) is a global
                }
            }
            // here we use substitution to solve T11 X w = c avoiding
the explicit formation of t11 inverse
            double[,] w = new double[n + 1, 2];
            w[1, 1] = QTb[1, 1] / T_encoded[1, 1];
            for (i = 2; i <= rank; i++)
            {
                for (j = 1; j <= i - 1; j++)
                {
                    w[i, 1] = w[i, 1] - T_encoded[i, j] * w[j, 1];
                }
                w[i, 1] = w[i, 1] + QTb[i, 1];
                w[i, 1] = w[i, 1] / T_encoded[i, i];
            }
            // form Z(T11)inv c = Zw
            m = T_encoded.GetLength(1) - 1;
            n = T_encoded.GetLength(0) - 1;
            for (j = n; j >= 1; j += -1)
            { // apply Q to w implicitly
```

```
                double[,] v = new double[m + 1, 2];
                v[j, 1] = 1;
                for (i = j + 1; i <= m; i++)
                {
                    v[i, 1] = T_encoded[j, i];
                }
                double[,] omega = new double[2, 2];
                omega = Matrix_Multiply(Transpose(w), v);
                for (i = 1; i <= m; i++)
                {
                    w[i, 1] = w[i, 1] - 1 * T_beta[j, 1] * omega[1, 1]
* v[i, 1]; // This beta(j) is a global
                }
            }
            double[,] tempw = new double[QR_encoded.GetLength(1) - 1 +
1, 2]; //  multiply by permutation
            for (i = 1; i <= QR_encoded.GetLength(1) - 1; i++)
            {
                tempw[QR_Encoded_P[i], 1] = w[i, 1];
            }
            if (null != done) done();
            return tempw;
        }


        public double[,]
Solve_Givens_complete_orthogonal_implicit(double[,] QR_encoded,
double[,] ZT, double[,] b)
        {
            // Golub/Van Loan p 250.
            int m = QR_encoded.GetLength(0) - 1;
            int n = QR_encoded.GetLength(1) - 1;
            int rank = ZT.GetLength(1) - 1;
            double[,] QTb = new double[m + 1, 2];
            double[] cs = new double[4];
            Array.Copy(b, QTb, b.Length);
            int i = 0, j = 0;
            for (j = 1; j <= rank; j++)
            { // determine c
                for (i = m; i >= j + 1; i += -1)
                {
                    if (QR_encoded[i, j] != 0)// Don't bother if rho is
0. It's just multiplying by one.
                    {
                        if (QR_encoded[i, j] == 1)
                        { //  this is the rho decoder section

                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(QR_encoded[i, j]) < 1)
                            {
                                cs[2] = 2 * QR_encoded[i, j];
```

```
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
                            else
                            {
                                cs[1] = 2 / QR_encoded[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute Q transpose b
                        double temp2 = 0;
                        temp1 = cs[1] * QTb[i - 1, 1] - cs[2] * QTb[i,
1];
                        temp2 = cs[2] * QTb[i - 1, 1] + cs[1] * QTb[i,
1];
                        QTb[i - 1, 1] = temp1;
                        QTb[i, 1] = temp2;
                    }
                }
            }
            // here we use substitution to solve T11 X w = c avoiding
the explicit formation of t11 inverse
            double[,] w = new double[n + 1, 2];
            w[1, 1] = QTb[1, 1] / ZT[1, 1];
            for (i = 2; i <= rank; i++)
            {
                for (j = 1; j <= i - 1; j++)
                {
                    w[i, 1] = w[i, 1] - ZT[j, i] * w[j, 1];
                }
                w[i, 1] = w[i, 1] + QTb[i, 1];
                w[i, 1] = w[i, 1] / ZT[i, i];
            }
            // Determine Zw=xls
            for (j = rank; j >= 1; j += -1)
            {
                for (i = j + 1; i <= n; i++)
                {
                    if (ZT[i, j] != 0)
                    { // Don't bother if rho is 0. It's just
multiplying by one.

                        if (ZT[i, j] == 1)
                        { //  this is the rho decoder section

                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(ZT[i, j]) < 1)
                            {
                                cs[2] = 2 * ZT[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
```

```
                            else
                            {
                                cs[1] = 2 / ZT[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0;
                        double temp2 = 0;
                        temp1 = cs[1] * w[i - 1, 1] + cs[2] * w[i, 1];
                        temp2 = cs[2] * -w[i - 1, 1] + cs[1] * w[i, 1];
                        w[i - 1, 1] = temp1;
                        w[i, 1] = temp2;
                    }
                }
            }
            double[,] tempw = new double[n + 1, 2]; //  multiply by
permutation
            for (i = 1; i <= n; i++)
            {
                tempw[QR_Encoded_P[i], 1] = w[i, 1];
            }
            if (null != done) done();
            return tempw;
        }
```

Let us look at our example using the complete orthogonalization approach:

$$A = \begin{bmatrix} 1 & 1.5 & 1 & 2 \\ 2 & 3 & 3 & 4 \\ 3 & 4.5 & 2 & 6 \\ 4 & 6 & 5 & 8 \\ 5 & 7.5 & 4 & 10 \end{bmatrix} \text{ and } b = \{12, 27, 33, 51, 57\}^T$$

Complete orthogonalization, Householder variant:

$$QR_{enc} = \begin{bmatrix} 14.8324 & 7.146519 & 11.1243 & 7.416198 \\ -0.311711 & 1.981735 & 0 & 0 \\ -0.4675666 & 0.973525 & 0 & 0 \\ -0.6234221 & -1.301281 & 0 & 0 \\ -0.7792776 & 0.8798804 & 0 & 0 \end{bmatrix}, \; P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$QRbeta = 0.86516$ and 0.4529731

$$ZT_{enc} = \begin{bmatrix} 21.20902 & -1.120736 & -1.744542 & -1.163028 \\ 0.6677584 & 1.865843 & 1.841821 & 1.227881 \end{bmatrix}$$

$Tbeta = 0.3006563$ and 0.3389833

Rank = 2

$x_{ls} = \{1.241379, 1.862069, 3, 2.482759\}^T$

Complete orthogonalization, Givens variant:

$$Q\mathrm{R}_{enc} = \begin{bmatrix} 14.8324 & 7.146519 & 11.1243 & 7.416198 \\ 14.8324 & -1.981735 & 0 & 0 \\ -7.348469 & -108 & 0 & 0 \\ 4.714045 & 3.550217 & 0 & 0 \\ -3.201562 & 3.901133 & 0 & 0 \end{bmatrix}, \; P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$ZT_{enc}^T = \begin{bmatrix} 21.20902 & -0.6677584 \\ 2.859824 & 1.865843 \\ -4.242599 & 5.711773 \\ -0.2773501 & 0 \end{bmatrix}$$

Rank = 2

$x_{ls} = \{1.241379, 1.862069, 3, 2.482759\}^T$

The Euclidian norm of our x vector is 4.49, compared to a norm of 5.41 for the QR with pivoting result. Note also that in this case each column of our coefficient matrix has a corresponding non zero multiplier in x.

5. Rank Determination, The Singular Value Decomposition, and Linear Solutions Using The Singular Value Decomposition

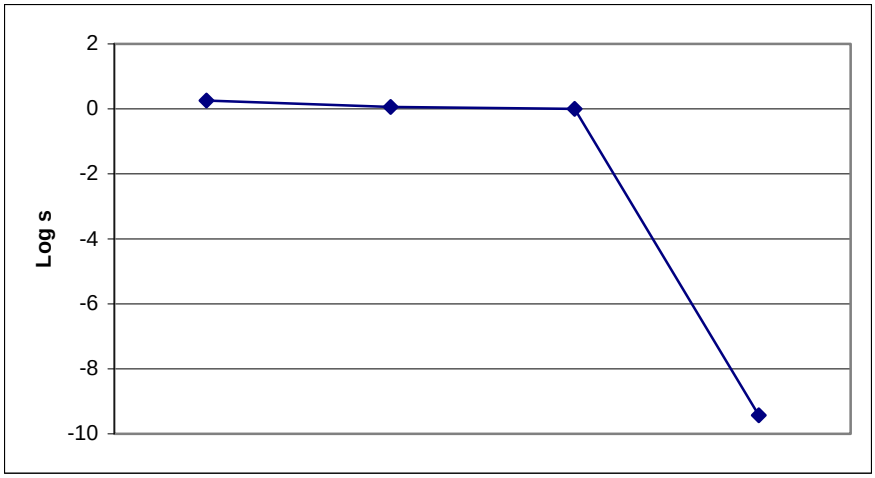
Re-envisioning Rank and Linear Independence

So far we have understood that rank is a function of column independence in a matrix, and that this independence is a quality that vectors do or do not have. It turns out that independence is not a black and white attribute but rather is characterized by shades of grey. Because rank is a function of independence, it also has a fuzzy definition. Two vectors are maximally independent when they are orthogonal (think perpendicular in Cartesian space). The dot product of maximally independent vectors is zero. As this condition degrades, vectors become increasingly dependent and matrices composed of those vectors become increasingly rank deficient. We seek a means of revealing rank in a matrix, and a means of obtaining solutions using this rank-revealing technique. The singular value decomposition (SVD) is the tool we will use. In general, the SVD is another complete orthogonal decomposition yielding a factorization of the form $U^T AV = \sum$ or $A = U \sum V^T$, where U^{mxm} is orthogonal, $\sum^{mxn}$ is diagonal, and V^{nxn} is orthogonal. It is by convention that sigma ($\sum^{mxn}$) has its diagonal elements expressed as positive values in descending order.

It is in the diagonal elements of sigma that the rank of A is revealed. Sigma for an orthogonal matrix (one that is maximally full rank) has ones for each of its diagonal elements. We can prepare a graph of sigma's diagonal elements for any given matrix A. Consider the matrix:

$$A = \begin{bmatrix} 0.1048285 & 0.9256937 & -0.2393573 & 0.6420752 \\ 0.4193139 & 0.2290376 & 1.054945 & 1.087337 \\ 0.7337994 & -0.4676185 & -0.2393573 & -0.6245721 \\ 0.5241424 & 0.286297 & -0.4609845 & -0.1238837 \end{bmatrix}$$

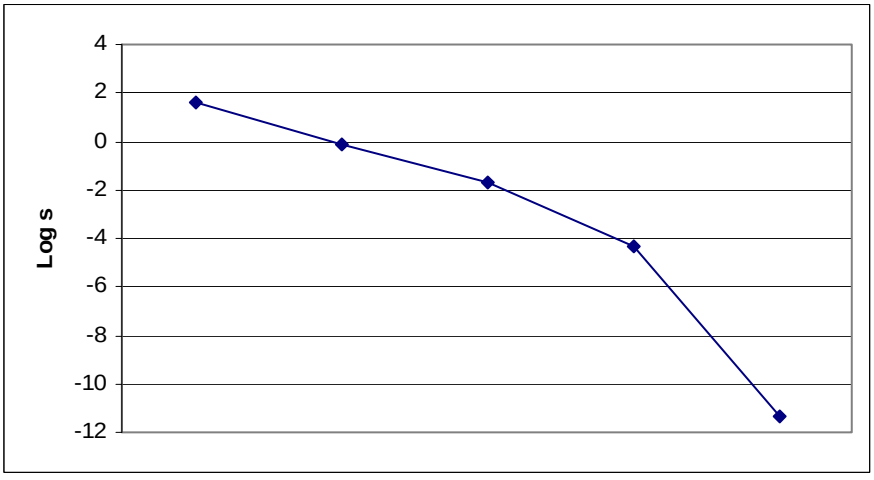
$$\sum = diag(1.83,\ 1.15,\ 1,\ 3.7E\text{-}10)$$



It is evident that this matrix has a rank of 3. However, it isn't always this clear. For example:

$$A = \begin{bmatrix} 9.65 & 11.22364 & 10.69638 & 10.65454 & 10.65005 \\ 7.93 & 9.443055 & 8.972282 & 8.93415 & 8.930041 \\ 7.9 & 9.411908 & 8.942204 & 8.904142 & 8.900042 \\ 3.02 & 4.267387 & 4.042351 & 4.022213 & 4.020022 \\ 3.7 & 4.999094 & 4.726512 & 4.70262 & 4.700026 \end{bmatrix}$$

$$\sum = diag(39,\ 0.69,\ 0.02,\ 4.4E-5,\ 5E-12)$$



This matrix is constructed as follows:

column2=column1 + column1 ↑ 0.2
column3=column1 + column1 ↑ 0.02
column4=column1 + column1 ↑ 0.002
column5=column1 + column1 ↑ 0.0002

Formally, the matrix should have full rank. Clearly the SVD provides information assisting in the appropriate assignment of rank based on application specifics.

Before we explore implemented algorithms for calculating the SVD we should discuss a certain amount of background. As promised in Chapter 1 we will be deliberately qualitative. Other texts are appropriate for intimately understanding the underlying mathematics. We must not lose track of our primary objective — to explore implementations.

Eigensystems

Consider the equation $Av = \lambda v$ where A is a square matrix, v is a non-zero vector, and λ is a scalar. It states quite simply that multiplying the vector by the matrix returns a vector that is a scalar multiple of the original vector. The pair, v and λ, form an *eigenpair* for the matrix A. For any given matrix $A \in \mathbb{R}^{mxm}$ there are m eigenpairs, although the eigenvalues may be equal in certain cases. The set of eigenvalues for a matrix is termed the matrix's *spectrum*. Although the spectrum is unique, the set of associated eigenvectors is not because any linearly dependent vector is also an eigenvector ($A2v = \lambda 2v,\ A3v = \lambda 3v,$ etc.). The fundamental eigenvector has a norm of unity ($v = v / \|v\|$).

We define the characteristic equation of A as $\text{determinant}(\lambda I - A) = 0$. We can use this formula for simple matrices to calculate eigenvalues. For example:

$$A = \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix},\ \lambda I = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix},\ \lambda I - A = \begin{bmatrix} \lambda - 1 & -2 \\ -4 & \lambda - 3 \end{bmatrix}$$

$$\det(\lambda I - A) = (\lambda - 1)(\lambda - 3) - 8 = \lambda^2 - 4\lambda - 5 = 0$$

Clearly, this quadratic equation has two roots, -1 and 5. Not so evident is the set of unit norm eigenvectors: (-0.7071, 0.7071) and (-0.4472, -0.8944). If we multiply them out we indeed see that:

$$\begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}\begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix} = -1\begin{bmatrix} -0.7071 \\ 0.7071 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix}\begin{bmatrix} -0.4472 \\ -0.8944 \end{bmatrix} = 5\begin{bmatrix} -0.4472 \\ -0.8944 \end{bmatrix}.$$

In Chapter 1 we briefly discussed complex numbers. Frequently, certain roots of the characteristic equation have an imaginary component, and as such are proper complex numbers. In fact, although we are selectively ignoring this, the matrices we deal with could themselves be composed of complex elements. We defined the complex conjugate of a complex number $c = a + bi$ as $c^* = a - bi$. A *conjugate matrix* of a matrix is one where each element in the original matrix has been replaced by its complex conjugate. The *conjugate transpose* of a matrix is the transpose of its conjugate matrix and is denoted A^H. An Hermitian Matrix is a square matrix with symmetry through the

principal diagonal such that $A = A^H$. An Hermitian matrix is said to be symmetric. For real matrices this definition reduces to $A = A^T$. Do not be put off by all of this — it is simply nomenclature. We will return to these symmetric matrices in a moment.

So far we have been implementing algorithms that produce a result in a direct, finite sequence of steps. Determining the set of eigenvectors for a matrix and determining the eigenvalues themselves — particularly when m is greater than 4 — requires the implementation of an iterative algorithm. This type of algorithm will (we hope) converge on the right answer through successive approximation. It turns out that for symmetric matrices, all of the eigenvalues are real and the matrix of unit-norm eigenvectors is orthogonal. We define the Schur Decomposition for symmetric matrices as $Q^T AQ = diag(\lambda_1, \lambda_2 ... \lambda_n)$, where all of the eigenvalues are real and Q is orthogonal. Also, the first column of Q and λ_1 form an eigenpair as do the second column of Q and λ_2 , etc. There are iterative procedures for obtaining this decomposition. Usually this is done in stages, beginning with a preliminary tridiagonalization of the matrix. A tridiagonal matrix has the form $A\begin{pmatrix} a_1 & b_1 & 0 \\ b_1 & \ddots & b_{m-1} \\ 0 & b_{m-1} & a_m \end{pmatrix}$. Given this form, a procedure called the Symmetric QR algorithm is used to iteratively obtain the final result.

The computation of eigenvalues and eigenvectors has value in many applications, most notably the solution to differential equations. Our objective, however, is to find the SVD of an *mxn* matrix which is not necessarily a square system. However, the iterative algorithms for solving eigensystems are the fundamental algorithms for computing the SVD. Recall from our discussion of the method of normal equations that A^TA is square. It is also symmetric, and it can be reduced to tridiagonal form and submitted to the symmetric QR algorithm. Recall too that the SVD of A is given by $U^T AV = \sum = diag(\sigma_1, \sigma_2, ..., \sigma_n)$, where σ_i are the positive singular values in descending order and U and V are orthogonal. It is further the case that $V^T(A^T A)V = diag(\sigma_1^2, \sigma_2^2, ..., \sigma_n^2)$ and $U^T(AA^T)U = diag(\sigma_1^2, \sigma_2^2, ..., \sigma_n^2, 0_{n+1}, ..., 0_m)$. Notice that these results have the conspicuous look of Schur decompositions for A^TA and AA^T. So, we could form A^TA, then use reduction to tridiagonal form followed by the QR algorithm to find V. We can then apply our QR with column pivoting procedure to AV to arrive at $UT(AV)P = R = \sum$. All we need to do at that point is organize the elements of $\sum$ so that they are positive and descending. Corresponding adjustments in U and V will of course need to be made.

That is about as far into the theory as were going to go here, except to say that certain enhancements to the procedure will be made. For example, we know that forming A^TA leads to a serious loss of information, so we will use a bidiagonalization procedure that will enable shortcutting the formation of A^TA and reducing it to tridiagonal form. Another enhancement involves the use of a convergence enhancing process called a Wilkinson shift. The processes and procedures underlying the algorithm we will implement are fascinating, and you are definitely encouraged to read more (see Golub/Van Loan, Chapter 8).

The Singular Value Decomposition

We will implement the Golub-Reinsch and Chan-R-SVD algorithms. Both will employ the Golub-Kahan step. This algorithm's strength is that it applies well to all matrices (although not necessarily with optimum efficiency), and it converges without exception. There are many other algorithms, and Bjorck has an excellent bibliography for researching these approaches.

It has been my approach in previous chapters to present factorization methods, then to present methods for solving Ax=b using those factorization techniques. Here we will have to break somewhat with this approach because some of the implicit techniques — which are fundamentally important here — only have meaning in the context of the least squares problem.

Solving Linear Systems with the SVD

When we compute the SVD of a matrix we begin by forming a bidiagonal factorization: $A = U_{bd} B V_{bd}^T$. We will then obtain the SVD of B, not A, giving us $B = U_\Sigma \Sigma V_\Sigma^T$, where $\Sigma = diag(\sigma_i)$. We define $U = U_{bd} U_\Sigma$ and $V = V_{bd} V_\Sigma$. It follows that the SVD of A is $A = U \Sigma V^T$.

For the system Ax=b, $Ax = (U \Sigma V^T)x = b.$ If we consider U and V to be sets of column vectors — $U = (u_i)$ and $V = (v_i)$ — then $xls = \sum_{i=1}^{rank} \frac{u_i^T b v_i}{\sigma_i}$.

Another way of approaching the solution employs a device called the *pseudo-inverse*. The pseudo-inverse, designated A^+, is given by $A^+ = V \Sigma^+ U^T$, where $\Sigma^+ = diag(1/\sigma_1, ..., 1/\sigma_r, 0_{r+1}, ..., 0_n)$. Here $x_{ls} = A^+ b$.

Both approaches require that we form $U^T b = \left(U_{bd} U_\Sigma\right)^T b$. At first it appears that we have a problem if we want to avoid forming U^{mxm} explicitly as we compute $U^T b$, because U^T is obtained through two transformations — with U_{bd} occurring first. However, $U^T b = \left(U_{bd} U_\Sigma\right)^T b = U_\Sigma^T U_{bd}^T b$, which is fortunate because we form the bidiagonalization transformations prior to forming the SVD transformations. So, in our implicit procedures you will see arguments pertaining to $U_{bd}^T b$ (`UbdTb`) and $U_\Sigma^T U_{bd}^T b$ (`UsvdT_UbdT_b`) being passed.

Bidiagonalization

Here we will factor an m x n matrix to bidiagonal form using Givens transformations. Householder transformations work equally well and theoretically are

faster. (The Golub-Reinsch Algorithm uses Householder orthogonalization, but the implementations given here use Givens orthogonalization because Givens technique is more adaptable to special systems.) The factorization will take the general form $A = U_{bd} BDV_{bd}$, where the bidiagonal matrix has the form:

$$BD = \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & c_{n-1} \\ \vdots & \ddots & \ddots & b_n \\ 0 & \cdots & \cdots & 0 \end{bmatrix}, \text{ and } U_{bd}^{mxm} \text{ as well as } V_{bd}^{nxn} \text{ are orthogonal.}$$

Obtaining this factorization involves orthogonal transformations to zero below the principal diagonal (accumulating U), interwoven with orthogonal transformations to zero row elements to the right of the super diagonal (accumulating V). Here is a subroutine that explicitly performs this task:

```
        public void Bidiagonalize_explicit_Givens(double[,]
Source_Matrix, ref double[,] BD, ref double[,] UBD, ref double[,] VBD)
        {
            int i = 0, j = 0, k = 0, m = 0, n = 0;
            if (Source_Matrix.GetLength(1) - 1 >
Source_Matrix.GetLength(0) - 1)
            {
                m = Source_Matrix.GetLength(1) - 1; // allow for
underdetermined system
            }
            else
            {
                m = Source_Matrix.GetLength(0) - 1;
            }
            n = Source_Matrix.GetLength(1) - 1;
            BD = new double[m + 1, n + 1];
            UBD = new double[m + 1, m + 1];
            VBD = new double[n + 1, n + 1];
            Array.Copy(Source_Matrix, BD, Source_Matrix.Length);
            for (i = 1; i <= m; i++)
            {
                UBD[i, i] = 1;
            }
            for (i = 1; i <= n; i++)
            {
                VBD[i, i] = 1;
            }
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(BD[i, j]) > Givens_Zero_Value)
                    { //  If it's already zero we won't zero it.
                        double[,] temp_R = new double[3, n - j + 1 +
1];
```

```
                        double[,] temp_U = new double[m + 1, 3];
                        double[] cs = new double[3];
                        cs = Givens(BD[i - 1, j], BD[i, j]);
                        // accumulate R = BD
                        for (k = j; k <= n; k++)
                        {
                            temp_R[1, -j + 1 + k] = cs[1] * BD[i - 1,
k] - cs[2] * BD[i, k];
                            temp_R[2, -j + 1 + k] = cs[2] * BD[i - 1,
k] + cs[1] * BD[i, k];
                        }
                        for (k = 1; k <= n - j + 1; k++)
                        {
                            BD[i - 1, j + k - 1] = temp_R[1, k];
                            BD[i, j + k - 1] = temp_R[2, k];
                        }
                        // accumulate UBD
                        for (k = 1; k <= m; k++)
                        {
                            temp_U[k, 1] = cs[1] * UBD[k, i - 1] -
cs[2] * UBD[k, i];
                            temp_U[k, 2] = cs[2] * UBD[k, i - 1] +
cs[1] * UBD[k, i];
                        }
                        for (k = 1; k <= m; k++)
                        {
                            UBD[k, i - 1] = temp_U[k, 1];
                            UBD[k, i] = temp_U[k, 2];
                        }
                    }
                }
                if (j <= n - 2)
                {
                    for (i = n; i >= j + 2; i += -1)
                    {
                        double[,] temp_R = new double[m - j + 1 + 1,
3];
                        double[,] temp_V = new double[n + 1, 3];
                        double[] cs = new double[3];
                        cs = Givens(BD[j, i - 1], BD[j, i]);
                        for (k = j; k <= m; k++)
                        {
                            temp_R[-j + 1 + k, 1] = cs[1] * BD[k, i -
1] - cs[2] * BD[k, i];
                            temp_R[-j + 1 + k, 2] = cs[2] * BD[k, i -
1] + cs[1] * BD[k, i];
                        }
                        for (k = 1; k <= m - j + 1; k++)
                        {
                            BD[j + k - 1, i - 1] = temp_R[k, 1];
                            BD[j + k - 1, i] = temp_R[k, 2];
                        }

                        for (k = 1; k <= n; k++)
                        {
                            temp_V[k, 1] = cs[1] * VBD[k, i - 1] -
cs[2] * VBD[k, i];
```

```
                            temp_V[k, 2] = cs[2] * VBD[k, i - 1] +
cs[1] * VBD[k, i];
                        }
                        for (k = 1; k <= n; k++)
                        {
                            VBD[k, i - 1] = temp_V[k, 1];
                            VBD[k, i] = temp_V[k, 2];
                        }
                    }
                }
            }
            if (null != done) done();
            for (i = 1; i <= n; i++)
            {
                for (k = i + 1; k <= m; k++)
                {
                    BD[k, i] = 0;
                }
            }
            for (k = 1; k <= n - 2; k++)
            {
                for (i = k + 2; i <= n; i++)
                {
                    BD[k, i] = 0;
                }
            }
        }
```

We can store the Givens rotators and avoid the explicit formation of U_{bd} with the following implicit version:

```
        public void Bidiagonalize_implicit_Givens(double[,]
Source_Matrix, ref double[,] bd_encoded, ref double[,] Vbd)
        {
            // Golub/Van Loan 5.4.2.  Does not overwrite Source_matrix.
            int i = 0, j = 0, k = 0, m = 0, n = 0;
            double rho = 0;
            if (Source_Matrix.GetLength(1) - 1 >
Source_Matrix.GetLength(0) - 1)
            {
                m = Source_Matrix.GetLength(1) - 1; // allow for
underdetermined system
            }
            else
            {
                m = Source_Matrix.GetLength(0) - 1;
            }
            n = Source_Matrix.GetLength(1) - 1;
            bd_encoded = new double[m + 1, n + 1];
            Vbd = new double[n + 1, n + 1];
            Array.Copy(Source_Matrix, bd_encoded,
Source_Matrix.Length);
            for (i = 1; i <= n; i++)
            {
                Vbd[i, i] = 1;
            }
```

```
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(bd_encoded[i, j]) >
Givens_Zero_Value)
                    { //  If it's already zero we won't zero it.

                        double[,] temp_R = new double[3, n - j + 1 +
1];
                        double[,] temp_U = new double[m + 1, 3];
                        double[] cs = new double[3];
                        cs = Givens(bd_encoded[i - 1, j], bd_encoded[i,
j]);
                        // accumulate R = bd_encoded
                        for (k = j; k <= n; k++)
                        {
                            temp_R[1, -j + 1 + k] = cs[1] *
bd_encoded[i - 1, k] - cs[2] * bd_encoded[i, k];
                            temp_R[2, -j + 1 + k] = cs[2] *
bd_encoded[i - 1, k] + cs[1] * bd_encoded[i, k];
                        }
                        for (k = 1; k <= n - j + 1; k++)
                        {
                            bd_encoded[i - 1, j + k - 1] = temp_R[1,
k];
                            bd_encoded[i, j + k - 1] = temp_R[2, k];
                        }
                        if (cs[1] == 0)
                        { // This is the rho encoder section

                            rho = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(cs[2]) <=
System.Math.Abs(cs[1]))
                            { // here we want to include "="

                                rho = System.Math.Sign(cs[1]) * cs[2] /
2;
                            }
                            else
                            {
                                rho = 2 * System.Math.Sign(cs[2]) /
cs[1];
                            }
                        }
                        bd_encoded[i, j] = rho;
                    }
                }
                if (j <= n - 2)
                {
                    for (i = n; i >= j + 2; i += -1)
                    {
                        double[,] temp_R = new double[m - j + 1 + 1,
3];
```

```
                        double[,] temp_V = new double[n + 1, 3];
                        double[] cs = new double[3];
                        cs = Givens(bd_encoded[j, i - 1], bd_encoded[j,
i]);
                        for (k = j; k <= m; k++)
                        {
                            temp_R[-j + 1 + k, 1] = cs[1] *
bd_encoded[k, i - 1] - cs[2] * bd_encoded[k, i];
                            temp_R[-j + 1 + k, 2] = cs[2] *
bd_encoded[k, i - 1] + cs[1] * bd_encoded[k, i];
                        }
                        for (k = 1; k <= m - j + 1; k++)
                        {
                            bd_encoded[j + k - 1, i - 1] = temp_R[k,
1];
                            bd_encoded[j + k - 1, i] = temp_R[k, 2];
                        }
                        for (k = 1; k <= n; k++)
                        {
                            temp_V[k, 1] = cs[1] * Vbd[k, i - 1] -
cs[2] * Vbd[k, i];
                            temp_V[k, 2] = cs[2] * Vbd[k, i - 1] +
cs[1] * Vbd[k, i];
                        }
                        for (k = 1; k <= n; k++)
                        {
                            Vbd[k, i - 1] = temp_V[k, 1];
                            Vbd[k, i] = temp_V[k, 2];
                        }
                    }
                }
            }
            if (null != done) done();
            for (k = 1; k <= n - 2; k++)
            {
                for (i = k + 2; i <= n; i++)
                {
                    bd_encoded[k, i] = 0;
                }
            }
        }
```

We can speed up the bidiagonalization process when m>>n with R-bidiagonalization (see Chan). With R bidiagonalization, we perform a preliminary QR factorization to do the sub-diagonal zeroing, then submit the upper *nxn* matrix (R_1) to the regular bidiagonalization method. U_{rbd} is the product of Q and the U from the latter bidiagonalization extended to an *mxm* matrix with the identity matrix and $V_{rbd}=V_{bd}$. Here is the implementation:

```
        public void R_Bidiagonalize_full(double[,] Source_Matrix, ref
double[,] RBD, ref double[,] URBD, ref double[,] VRBD)
        {
            // Chan p.72 - 83 Overwrites Source_Matrix. Uses
Householder QR Simple and is not
            // usable with underdetermined systems as writen.
```

```
            int m = 0, n = 0, i = 0, j = 0, k = 0;
            m = Source_Matrix.GetLength(0) - 1;
            n = Source_Matrix.GetLength(1) - 1;
            RBD = new double[m + 1, n + 1];
            URBD = new double[m + 1, m + 1];
            // QR Factor
            double[,] Q = null;
            double[,] R = null;
            Q = Householder_QR_Simple(Source_Matrix);
            R = Source_Matrix;
            // extract nxn square of R
            double[,] R1 = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    R1[i, j] = R[i, j];
                }
            }
            // bidiagonalize that square
            double[,] BD = null;
            double[,] UBD = null;
            Bidiagonalize_explicit_Givens(R1, ref BD, ref UBD, ref
VRBD);
            Array.Copy(BD, RBD, BD.Length); // conserves length of
destination array
            for (i = 1; i <= m; i++)
            {
                for (k = 1; k <= n; k++)
                {
                    for (j = 1; j <= n; j++)
                    {
                        URBD[i, j] = Q[i, k] * UBD[k, j] + URBD[i, j];
                    }
                }
            }
            for (i = 1; i <= m; i++)
            {
                for (j = n + 1; j <= m; j++)
                {
                    URBD[i, j] = Q[i, j];
                }
            }
            for (i = 1; i <= n; i++)
            {
                for (k = i + 1; k <= m; k++)
                {
                    RBD[k, i] = 0;
                }
            }
            for (k = 1; k <= n - 2; k++)
            {
                for (i = k + 2; i <= n; i++)
                {
                    RBD[k, i] = 0;
                }
            }
```

```
            if (null != done) done();
        }
```

The next method takes b as an argument, then implicitly forms and returns $U_{bd}^{T} b$ (`UbdTb`). It is non-reusable.

```
        public void R_Bidiagonalize_implicit(double[,] Source_Matrix,
double[,] b, ref double[,] RBD, ref double[,] Vbd, ref double[,] UbdTb)
        {
            // Chan p.72 - 83. Note that Ubd=QUr(extended).
UbdTb=QUr(extended)Tb=UrextendedT X QTb. Immediate formation of UbdTb.
            // Overwrites Source_matrix. Do not use for underdetermined
systems.
            int i = 0, j = 0, k = 0, m = 0, n = 0;
            m = Source_Matrix.GetLength(0) - 1;
            n = Source_Matrix.GetLength(1) - 1;
            RBD = new double[m + 1, n + 1];
            Vbd = new double[n + 1, n + 1];
            UbdTb = new double[n + 1, 2];
            // perform QR encoded
            Source_Matrix = Givens_QR_Simple_encoded(Source_Matrix);
            // break out R1
            double[,] R1 = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                for (j = i; j <= n; j++)
                {
                    R1[i, j] = Source_Matrix[i, j];
                }
            }
            // make QTb
            double[,] QTb = new double[m + 1, 2];
            Array.Copy(b, QTb, b.Length);
            double[] cs = new double[4];
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (Source_Matrix[i, j] != 0)
                    { // Don't bother if rho is 0. It's just
multiplying by one.

                        if (Source_Matrix[i, j] == 1)
                        { //  this is the rho decoder section

                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(Source_Matrix[i, j]) <
1)
                            {
                                cs[2] = 2 * Source_Matrix[i, j];
```

```
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
                            else
                            {
                                cs[1] = 2 / Source_Matrix[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute QTb
                        double temp2 = 0;
                        temp1 = cs[1] * QTb[i - 1, 1] - cs[2] * QTb[i,
1];
                        temp2 = cs[2] * QTb[i - 1, 1] + cs[1] * QTb[i,
1];
                        QTb[i - 1, 1] = temp1;
                        QTb[i, 1] = temp2;
                    }
                }
            }
            // bidiagonalize r1
            double[,] UBD = null;
            Bidiagonalize_explicit_Givens(R1, ref RBD, ref UBD, ref
Vbd);
            // create a matrix that holds U in the upper nxn square and
extends to mxm with the identity matrix
            // Ubd=Q X (Urextended) so UbdTb=(Urextended)Transpose X
QTb
            for (i = 1; i <= n; i++)
            {
                for (k = 1; k <= n; k++)
                {
                    UbdTb[i, 1] = UBD[k, i] * QTb[k, 1] + UbdTb[i, 1];
// Ur(k,i)-Ur(i,k)T
                }
            }
            if (null != done) done();
              }
```

For many systems we do not need a full Q, because the first n columns of U_{rbd} will suffice. The following procedure is a U_1 truncated R-bidiagonalization:

```
        public void R_Bidiagonalize_U1(double[,] Source_Matrix, ref
double[,] RBD, ref double[,] URBD, ref double[,] VRBD)
        {
            // Chan p.72 - 83. Overwrites Source_Matrix. Not for use
with undetermined systems
            int m = 0, n = 0, i = 0, j = 0, k = 0;
            m = Source_Matrix.GetLength(0) - 1;
            n = Source_Matrix.GetLength(1) - 1;
            URBD = new double[m + 1, n + 1];
            // QR Factor
            double[,] Q = null;
            double[,] R = null;
            Q = Householder_QR_Simple(Source_Matrix);
```

```
            R = Source_Matrix;
            // extract nxn square of R
            double[,] R1 = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    R1[i, j] = R[i, j];
                }
            }
            // bidiagonalize that square
            double[,] UBD = null;
            Bidiagonalize_explicit_Givens(R1, ref RBD, ref UBD, ref
VRBD);
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    for (k = 1; k <= n; k++)
                    {
                        URBD[i, k] = Q[i, j] * UBD[j, k] + URBD[i, k];
                    }
                }
            }
            for (i = 1; i <= n; i++)
            {
                for (k = i + 1; k <= n; k++)
                {
                    RBD[k, i] = 0;
                }
            }
            for (k = 1; k <= n - 2; k++)
            {
                for (i = k + 2; i <= n; i++)
                {
                    RBD[k, i] = 0;
                }
            }
            if (null != done) done();
        }
```

The next two methods are reusable implicit R-bidiagonalization implementations. The first method implicitly forms the Q but explicitly forms R_1, the intermediate U_{r1} which is the U portion of the bidiagonalization of R, and V_{bd}:

```
        public void R_Bidiagonalize_implicit_1(double[,] Source_Matrix,
ref double[,] QR, ref double[,] R1BD, ref double[,] UR1, ref double[,]
Vbd)
        {
            // Chan p.72 - 83. Note that Ubd=QUr(extended). Implicit
QR, explicit R_bd. Re-usable factorization
            // Do not use for underdetermined systems.
            int i = 0, j = 0, m = 0, n = 0;
            m = Source_Matrix.GetLength(0) - 1;
            n = Source_Matrix.GetLength(1) - 1;
            QR = new double[m + 1, n + 1];
```

```
            // perform QR encoded
            Array.Copy(Source_Matrix, QR, Source_Matrix.Length);
            QR = Givens_QR_Simple_encoded(QR);
            // break out R1
            double[,] R1 = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                for (j = i; j <= n; j++)
                {
                    R1[i, j] = QR[i, j];
                }
            }
            // bidiagonalize r1 explicitly
            Bidiagonalize_explicit_Givens(R1, ref R1BD, ref UR1, ref
Vbd);
            if (null != done) done();
              }
```

The second procedure takes it to the next level, returning an encoded QR, an encoded R1bd, and Vbd:

```
        public void R_Bidiagonalize_implicit_2(double[,] Source_Matrix,
ref double[,] QR, ref double[,] R1BD_enc, ref double[,] Vbd)
        {
            // Chan p.72 - 83. Note that Ubd=QUr(extended). Implicit
QR, implicit R BD. Re-usable factorization
            // Do not use for underdetermined systems.
            int i = 0, j = 0, m = 0, n = 0;
            m = Source_Matrix.GetLength(0) - 1;
            n = Source_Matrix.GetLength(1) - 1;
            QR = new double[m + 1, n + 1];
            // perform QR encoded
            Array.Copy(Source_Matrix, QR, Source_Matrix.Length);
            QR = Givens_QR_Simple_encoded(QR);
            // break out R1
            double[,] R1 = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                for (j = i; j <= n; j++)
                {
                    R1[i, j] = QR[i, j];
                }
            }
            // bidiagonalize r1 implicitly
            Bidiagonalize_implicit_Givens(R1, ref R1BD_enc, ref Vbd);
            if (null != done) done();
              }
```

Three of the methods given above provide a reusable implicit factorization. To realize the overall objective of implicit formation of $U^T b$, we now have to commit to a specified b vector and form $U_{bd}^T b$. There are three methods we must address:

Bidiagonalize_implicit_Givens
R_Bidiagonalize_implicit_1
R_Bidiagonalize_implicit_2

For Bidiagonalize_implicit_Givens we have:

```
        public double[,] Form_UbdTb_Givens(double[,] Encoded_BD_Matrix,
double[,] b_vector)
        {
            // Intermediate function for implicit SVD solution Note:
(Ubd X Usvd)T = UsvdT X UbdT
            int i = 0, j = 0;
            int m = Encoded_BD_Matrix.GetLength(0) - 1;
            int n = Encoded_BD_Matrix.GetLength(1) - 1;
            double[] cs = new double[3];
            double[,] UbdTb = new double[m + 1, 2];
            Array.Copy(b_vector, UbdTb, b_vector.Length);
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (Encoded_BD_Matrix[i, j] != 0)
                    { // Don't bother if rho is 0. It's just
multiplying by one.

                        if (Encoded_BD_Matrix[i, j] == 1)
                        { //  this is the rho decoder section

                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(Encoded_BD_Matrix[i,
j]) < 1)
                            {
                                cs[2] = 2 * Encoded_BD_Matrix[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
                            else
                            {
                                cs[1] = 2 / Encoded_BD_Matrix[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute QTb
                        double temp2 = 0;
                        temp1 = cs[1] * UbdTb[i - 1, 1] - cs[2] *
UbdTb[i, 1];
                        temp2 = cs[2] * UbdTb[i - 1, 1] + cs[1] *
UbdTb[i, 1];
                        UbdTb[i - 1, 1] = temp1;
                        UbdTb[i, 1] = temp2;
                    }
                }
            }
            if (null != done) done();
            return UbdTb;
```

```
        }
```

For R_Bidiagonalize_implicit_1 we have:

```
        public double[,] Form_UbdTb_implicit_R_BD_1(double[,]
Q_encoded, double[,] Ur1, double[,] b)
        {
            // use with encoded Q and explicit U r1
            int m = Q_encoded.GetLength(0) - 1;
            int n = Q_encoded.GetLength(1) - 1;
            int i = 0, j = 0, k = 0;
            // make(QTb)
            double[,] QTb = new double[m + 1, 2];
            Array.Copy(b, QTb, b.Length);
            double[] cs = new double[4];
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (Q_encoded[i, j] != 0)
                    { // Don't bother if rho is 0. It's just
multiplying by one.

                        if (Q_encoded[i, j] == 1)
                        { //  this is the rho decoder section

                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(Q_encoded[i, j]) < 1)
                            {
                                cs[2] = 2 * Q_encoded[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
                            else
                            {
                                cs[1] = 2 / Q_encoded[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute QTb
                        double temp2 = 0;
                        temp1 = cs[1] * QTb[i - 1, 1] - cs[2] * QTb[i,
1];
                        temp2 = cs[2] * QTb[i - 1, 1] + cs[1] * QTb[i,
1];
                        QTb[i - 1, 1] = temp1;
                        QTb[i, 1] = temp2;
                    }
                }
            }
            double[,] UbdTb = new double[n + 1, 2];
```

```
            // create a matrix that holds U in the upper nxn square and
extends to mxm with the identity matrix
            // Ubd=Q X (Urextended) so UbdTb=(Urextended)Transpose X
QTb
            for (i = 1; i <= n; i++)
            {
                for (k = 1; k <= n; k++)
                {
                    UbdTb[i, 1] = Ur1[k, i] * QTb[k, 1] + UbdTb[i, 1];
// Ur(k,i)-Ur(i,k)T
                }
            }
            double[,] result = new double[n + 1, 2];
            for (i = 1; i <= n; i++)
            {
                result[i, 1] = UbdTb[i, 1];
            }
            if (null != done) done();
            return result;
        }
```

And, for R_Bidiagonalize_implicit_2 we have:

```
        public double[,] Form_UbdTb_implicit_R_BD_2(double[,]
Q_encoded, double[,] R1bd_enc, double[,] b)
        {
            // Use with implicit QR, implicit R1_bd
            int m = Q_encoded.GetLength(0) - 1;
            int n = Q_encoded.GetLength(1) - 1;
            int i = 0, j = 0;
            // make(QTb)
            double[,] QTb = new double[m + 1, 2];
            Array.Copy(b, QTb, b.Length);
            double[] cs = new double[4];
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (Q_encoded[i, j] != 0)
                    { // Don't bother if rho is 0. It's just
multiplying by one.

                        if (Q_encoded[i, j] == 1)
                        { //  this is the rho decoder section

                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(Q_encoded[i, j]) < 1)
                            {
                                cs[2] = 2 * Q_encoded[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
```

```
                            else
                            {
                                cs[1] = 2 / Q_encoded[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute QTb
                        double temp2 = 0;
                        temp1 = cs[1] * QTb[i - 1, 1] - cs[2] * QTb[i,
1];
                        temp2 = cs[2] * QTb[i - 1, 1] + cs[1] * QTb[i,
1];
                        QTb[i - 1, 1] = temp1;
                        QTb[i, 1] = temp2;
                    }
                }
            }
            double[,] UbdTb = new double[n + 1, 2];
            Array.Copy(QTb, UbdTb, UbdTb.Length);

            for (j = 1; j <= n; j++)
            {
                for (i = n; i >= j + 1; i += -1)
                {
                    if (R1bd_enc[i, j] != 0)
                    { // Don't bother if rho is 0. It's just
multiplying by one.

                        if (R1bd_enc[i, j] == 1)
                        { //  this is the rho decoder section

                            cs[1] = 0;
                            cs[2] = 1;
                        }
                        else
                        {
                            if (System.Math.Abs(R1bd_enc[i, j]) < 1)
                            {
                                cs[2] = 2 * R1bd_enc[i, j];
                                cs[1] = Math.Pow((1 - (cs[2] * cs[2])),
0.5);
                            }
                            else
                            {
                                cs[1] = 2 / R1bd_enc[i, j];
                                cs[2] = Math.Pow((1 - (cs[1] * cs[1])),
0.5);
                            }
                        }
                        double temp1 = 0; // compute QTb
                        double temp2 = 0;
                        temp1 = cs[1] * UbdTb[i - 1, 1] - cs[2] *
UbdTb[i, 1];
                        temp2 = cs[2] * UbdTb[i - 1, 1] + cs[1] *
UbdTb[i, 1];
                        UbdTb[i - 1, 1] = temp1;
```

```
                        UbdTb[i, 1] = temp2;
                    }
                }
            }
            // create a matrix that holds U in the upper nxn square and
extends to mxm with the identity matrix
            // Ubd=Q X (Urextended) so UbdTb=(Urextended)Transpose X
QTb
            double[,] result = new double[n + 1, 2];
            for (i = 1; i <= n; i++)
            {
                result[i, 1] = UbdTb[i, 1];
            }
            if (null != done) done();
            return result;
        }
```

There are certainly many other variants of the procedures given above. There are also cases where only the diagonal vector of singular values is of interest, so we can eliminate accumulation and storage of the U and V matrices with substantial savings in processing overhead and storage. Also, we are using an entire n x n array to store the bidiagonal matrix. Doing this makes the implementation substantially easier to follow than the more efficient practice of storing this matrix as two vectors.

The SVD Step

The underlying algorithms for the methods in this section are provided in Golub/Van Loan, Chapter 8, pp 454–456. There are two algorithms: Algorithm 8.6.2, which is the SVD algorithm itself, and Algorithm 8.6.1, which is a subprocedure used by Algorithm 8.6.2.

The implementations provided for Algorithm 8.6.2 (there are two, one explicit and one implicit) append a rank determining step based on a global SVD_Rank_Determination_Threshold. The actual rank determination uses the product of this value and the Frobenius norm of $\sum$ (vector 2 norm of $diag(\sum)$) to establish rank. A value of 0.000001 rejects values below 1 part per million of the Frobenius norm of $\sum$. Later, in applying the SVD to obtaining linear solutions using the pseudo-inverse approach, we will have a chance to adjust this parameter and re-evaluate rank.

For all explicit bidiagonalization results we can obtain $U_{\sum}$, $V_{\sum}$, and an unorganized (not necessarily positive and descending) set of singular values $\sum$ given a bidiagonal matrix using the following method:

```
        public void Algo862_explicit(double[,]
bidiagonalized_sourse_array, ref double[,] Usvd, ref double[,] Sigma,
ref double[,] Vsvd)
        {
            // Golub/Van Loan 8.6.2. Does not overwrite Source_matrix
            int m = bidiagonalized_sourse_array.GetLength(0) - 1;
            int n = bidiagonalized_sourse_array.GetLength(1) - 1;
            int q = 0, p = 0, h = 0, i = 0, j = 0, k = 0;
```

```
            Usvd = new double[m + 1, m + 1];
            Sigma = new double[n + 1, n + 1];
            Vsvd = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                j = i; // copy BD to Sigma
                while (j < n)
                {
                    Sigma[i, j] = bidiagonalized_sourse_array[i, j];
                    Sigma[i, j + 1] = bidiagonalized_sourse_array[i, j
+ 1];
                    j = j + 1;
                }
                Sigma[i, j] = bidiagonalized_sourse_array[i, j];
            }
            for (i = 1; i <= n; i++)
            {
                Vsvd[i, i] = 1;
            }
            for (i = 1; i <= m; i++)
            {
                Usvd[i, i] = 1;
            }
            while (q < n)
            {
                // Set bi,i+1 to zero...
                for (i = 1; i <= n - 1; i++)
                {
                    if (System.Math.Abs(Sigma[i, i + 1]) <=
Machine_Error * (System.Math.Abs(Sigma[i, i]) + System.Math.Abs(Sigma[i
+ 1, i + 1])))
                    {
                        Sigma[i, i + 1] = 0;
                    }
                }
                // find the largest Q
                q = 0;
                for (i = n; i >= 1; i += -1)
                {
                    if (Sigma[i - 1, i] == 0)
                    {
                        q = q + 1;
                    }
                    else
                    {
                        break;
                    }
                }
                // and smallest P....
                p = n - q;
                for (i = n - q; i >= 2; i += -1)
                {
                    if (Sigma[i - 1, i] != 0)
                    {
                        p = p - 1;
                    }
                    else
```

```
                    {
                        break;
                    }
                }
                if (q < n)
                {
                    // if any diag on b22 is zero then zero the super
                    bool FoundZero = false;
                    for (h = p; h <= n - q; h++)
                    {
                        for (i = p; i <= n - q; i++)
                        {
                            if (Sigma[i, i] == 0)
                            {
                                Sigma[i - 1, i] = 0;
                                FoundZero = true;
                            }
                        }
                        if (FoundZero == true)
                        {
                            goto L1; // Here we use the unthinkable
goto statement. I think the code is clearer with it.
                        }
                        else
                        {
                            // apply algo 861 to B22
                            double[,] b22 = new double[n - p - q + 1 +
1, n - p - q + 1 + 1]; // form it
                            for (j = 1; j <= n - p - q + 1; j++)
                            {
                                for (k = 1; k <= n - p - q + 1; k++)
                                {
                                    b22[j, k] = Sigma[j + p - 1, k + p
- 1];
                                }
                            }
                            Algo861_explicit(b22, n, p, q, Usvd, Vsvd);
                            for (j = 1; j <= n - p - q + 1; j++)
                            { // sub it back in

                                for (k = 1; k <= n - p - q + 1; k++)
                                {
                                    Sigma[j + p - 1, k + p - 1] =
b22[j, k];
                                }
                            }
                        }
                    }
                }
            L1: { }
            }
            // Assign(rank)
            double FnormS = Frobenius_norm(Sigma);
            SVDRANK = n;
            for (i = 1; i <= n; i++)
            {
```

```
                if (System.Math.Abs(Sigma[i, i]) <
SVD_Rank_Determination_Threshold * FnormS)
                {
                    SVDRANK = SVDRANK - 1;
                }
            }
            if (null != done) done();
              }
```

This method uses the following subprocedures:

```
        private void Algo861_explicit(double[,] B22, int n, int p, int
q, double[,] U, double[,] V)
        {
            // Golub/Van Loan 8.6.1
            int i = 0, k = 0, npq = 0;
            npq = n - p - q + 1;
            // create certain elements of T = B22transpose * b22
implicitly
            double[,] TTT = new double[3, 3];
            TTT[1, 1] = (Math.Pow(B22[npq - 1, npq - 1], 2));
            TTT[1, 2] = B22[npq - 1, npq - 1] * B22[npq - 1, npq];
            TTT[2, 1] = TTT[1, 2];
            TTT[2, 2] = (Math.Pow(B22[npq, npq], 2)) +
(Math.Pow(B22[npq - 1, npq], 2));
            // create u = eigenvalue of trailing 2x2 submatrix of T
that is closer to tnn
            double d = (TTT[1, 1] - TTT[2, 2]) / 2;
            double u1 = TTT[2, 2] - (Math.Pow(TTT[2, 1], 2)) / (d +
(System.Math.Sign(d) * (Math.Pow(((Math.Pow(d, 2)) + (Math.Pow(TTT[2,
1], 2))), 0.5))));
            double y = (Math.Pow(B22[1, 1], 2)) - u1;
            double z = B22[1, 1] * B22[1, 2];
            for (k = 1; k <= npq - 1; k++)
            {
                // find c and s such that |y z||s/-s s/c|=|* 0|
                double[] cs = new double[4];
                Array.Copy(Givens(y, z), cs, cs.Length);
                // accumulate Vsigma TempV=TempVG
                double[,] summer = new double[V.GetLength(0) - 1 + 1,
3];
                for (i = 1; i <= n; i++)
                {
                    summer[i, 1] = cs[1] * V[i, k + p - 1] - cs[2] *
V[i, k + p];
                    summer[i, 2] = cs[2] * V[i, k + p - 1] + cs[1] *
V[i, k + p];
                }
                for (i = 1; i <= n; i++)
                {
                    V[i, k + p - 1] = summer[i, 1];
                    V[i, k + p] = summer[i, 2];
                }
                // apply givins transformation to B22 B22=TempB22G
                summer = new double[B22.GetLength(0) - 1 + 1, 3];
                for (i = 1; i <= B22.GetLength(0) - 1; i++)
                {
```

```
                    summer[i, 1] = cs[1] * B22[i, k] - cs[2] * B22[i, k
+ 1];
                    summer[i, 2] = cs[2] * B22[i, k] + cs[1] * B22[i, k
+ 1];
                }
                for (i = 1; i <= B22.GetLength(0) - 1; i++)
                {
                    B22[i, k] = summer[i, 1];
                    B22[i, k + 1] = summer[i, 2];
                }
                y = B22[k, k];
                z = B22[k + 1, k];
                // determine c&s .. |c/-s s/c|T * |y/z| = |* 0|
                cs = new double[3];
                Array.Copy(Givens(y, z), cs, cs.Length);
                // accumulate U TempU=TempUG
                summer = new double[n + 1, 3];
                for (i = 1; i <= n; i++)
                {
                    summer[i, 1] = +cs[1] * U[i, k + p - 1] - cs[2] *
U[i, k + p];
                    summer[i, 2] = +cs[2] * U[i, k + p - 1] + cs[1] *
U[i, k + p];
                }
                for (i = 1; i <= n; i++)
                {
                    U[i, k + p - 1] = summer[i, 1];
                    U[i, k + p] = summer[i, 2];
                }
                // 'apply givins transformation to B22 B22
B22=GTTempB22
                summer = new double[3, B22.GetLength(1) - 1 + 1];
                for (i = 1; i <= B22.GetLength(1) - 1; i++)
                {
                    summer[1, i] = cs[1] * B22[k, i] - cs[2] * B22[k +
1, i];
                    summer[2, i] = cs[2] * B22[k, i] + cs[1] * B22[k +
1, i];
                }
                for (i = 1; i <= B22.GetLength(1) - 1; i++)
                {
                    B22[k, i] = summer[1, i];
                    B22[k + 1, i] = summer[2, i];
                }
                if (k < npq - 1)
                {
                    y = B22[k, k + 1];
                    z = B22[k, k + 2];
                }
            }
        }

        public double Frobenius_norm(double[,] Source_Matrix)
        {
            int i = 0, j = 0;
            int m = Source_Matrix.GetLength(0) - 1;
```

```
            int n = Source_Matrix.GetLength(1) - 1;
            double result = 0;
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    result = result + (Math.Pow(Source_Matrix[i, j],
2));
                }
            }
            result = Math.Pow(result, 0.5);
            return result;
        }
```

The following two methods are analogs of the above methods in which transformations are applied to $U_{bd}^T b$ to form $U_\Sigma^T U_{bd}^T b = \left(U_{bd} U_\Sigma\right)^T b = U^T b$.

```
        public void Algo862_implicit(double[,]
bidiagonalized_sourse_array, double[,] UbdTb, ref double[,]
UsvdT_UbdT_b, ref double[,] sigma, ref double[,] Vsvd)
        {
            // Golub/Van Loan 8.6.2. Does not overwrite Source_matrix
            int m = bidiagonalized_sourse_array.GetLength(0) - 1;
            int n = bidiagonalized_sourse_array.GetLength(1) - 1;
            int q = 0, p = 0, h = 0, i = 0, j = 0, k = 0;
            UsvdT_UbdT_b = new double[m + 1, 2];
            Array.Copy(UbdTb, UsvdT_UbdT_b, UbdTb.Length);
            sigma = new double[n + 1, n + 1];
            Vsvd = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                j = i; // copy BD to SVD
                while (j < n)
                {
                    sigma[i, j] = bidiagonalized_sourse_array[i, j];
                    sigma[i, j + 1] = bidiagonalized_sourse_array[i, j
+ 1];
                    j = j + 1;
                }
                sigma[i, j] = bidiagonalized_sourse_array[i, j];
            }
            for (i = 1; i <= n; i++)
            {
                Vsvd[i, i] = 1;
            }
            while (q < n)
            {
                // Set bi,i+1 to zero...
                for (i = 1; i <= n - 1; i++)
                {
                    if (System.Math.Abs(sigma[i, i + 1]) <=
Machine_Error * (System.Math.Abs(sigma[i, i]) + System.Math.Abs(sigma[i
+ 1, i + 1])))
                    {
                        sigma[i, i + 1] = 0;
```

```
                    }
                }
                // find the largest Q
                q = 0;
                for (i = n; i >= 1; i += -1)
                {
                    if (sigma[i - 1, i] == 0)
                    {
                        q = q + 1;
                    }
                    else
                    {
                        break;
                    }
                }
                // and smallest P....
                p = n - q;
                for (i = n - q; i >= 2; i += -1)
                {
                    if (sigma[i - 1, i] != 0)
                    {
                        p = p - 1;
                    }
                    else
                    {
                        break;
                    }
                }
                if (q < n)
                {
                    // if any diag on b22 is zero then zero the super
                    bool FoundZero = false;
                    for (h = p; h <= n - q; h++)
                    {
                        for (i = p; i <= n - q; i++)
                        {
                            if (sigma[i, i] == 0)
                            {
                                sigma[i - 1, i] = 0;
                                FoundZero = true;
                            }
                        }
                        if (FoundZero == true)
                        {
                            goto L1; // Here we use the unthinkable
goto statement. I think the code is clearer with it.
                        }
                        else
                        {
                            // apply algo 861 to B22
                            double[,] b22 = new double[n - p - q + 1 +
1, n - p - q + 1 + 1]; // form it
                            for (j = 1; j <= n - p - q + 1; j++)
                            {
                                for (k = 1; k <= n - p - q + 1; k++)
                                {
```

```
                                    b22[j, k] = sigma[j + p - 1, k + p
- 1];
                                }
                            }
                            Algo861_implicit(b22, n, p, q,
UsvdT_UbdT_b, Vsvd);
                            for (j = 1; j <= n - p - q + 1; j++)
                            { // sub it back in

                                for (k = 1; k <= n - p - q + 1; k++)
                                {
                                    sigma[j + p - 1, k + p - 1] =
b22[j, k];
                                }
                            }
                        }
                    }
                }
            L1: { }
            }
            // Assign(rank)
            double FnormS = Frobenius_norm(sigma);
            SVDRANK = n;
            for (i = 1; i <= n; i++)
            {
                if (System.Math.Abs(sigma[i, i]) <
SVD_Rank_Determination_Threshold * FnormS)
                {
                    SVDRANK = SVDRANK - 1;
                }
            }
            if (null != done) done();
        }


        private void Algo861_implicit(double[,] B22, int n, int p, int
q, double[,] U, double[,] V)
        {
            // Golub/Van Loan 8.6.1
            int i = 0, k = 0, npq = 0;
            npq = n - p - q + 1;
            // create certain elements of T = B22transpose * b22
implicitly
            double[,] TTT = new double[3, 3];
            TTT[1, 1] = (Math.Pow(B22[npq - 1, npq - 1], 2));
            TTT[1, 2] = B22[npq - 1, npq - 1] * B22[npq - 1, npq];
            TTT[2, 1] = TTT[1, 2];
            TTT[2, 2] = (Math.Pow(B22[npq, npq], 2)) +
(Math.Pow(B22[npq - 1, npq], 2));
            // create u = eigenvalue of trailing 2x2 submatrix of T
that is closer to tnn
            double d = (TTT[1, 1] - TTT[2, 2]) / 2;
            double u1 = TTT[2, 2] - (Math.Pow(TTT[2, 1], 2)) / (d +
(System.Math.Sign(d) * (Math.Pow(((Math.Pow(d, 2)) + (Math.Pow(TTT[2,
1], 2))), 0.5))));
            double y = (Math.Pow(B22[1, 1], 2)) - u1;
            double z = B22[1, 1] * B22[1, 2];
```

```
            for (k = 1; k <= npq - 1; k++)
            {
                // find c and s such that |y z||s/-s s/c|=|* 0|
                double[] cs = new double[4];
                Array.Copy(Givens(y, z), cs, cs.Length);
                // accumulate Vsigma TempV=TempVG
                double[,] summer = new double[V.GetLength(0) - 1 + 1,
3];
                for (i = 1; i <= n; i++)
                {
                    summer[i, 1] = cs[1] * V[i, k + p - 1] - cs[2] *
V[i, k + p];
                    summer[i, 2] = cs[2] * V[i, k + p - 1] + cs[1] *
V[i, k + p];
                }
                for (i = 1; i <= n; i++)
                { // p + 1 To n

                    V[i, k + p - 1] = summer[i, 1];
                    V[i, k + p] = summer[i, 2];
                }
                // apply givins transformation to B22 B22=TempB22G
                summer = new double[B22.GetLength(0) - 1 + 1, 3];
                for (i = 1; i <= B22.GetLength(0) - 1; i++)
                {
                    summer[i, 1] = cs[1] * B22[i, k] - cs[2] * B22[i, k
+ 1];
                    summer[i, 2] = cs[2] * B22[i, k] + cs[1] * B22[i, k
+ 1];
                }
                for (i = 1; i <= B22.GetLength(0) - 1; i++)
                {
                    B22[i, k] = summer[i, 1];
                    B22[i, k + 1] = summer[i, 2];
                }
                y = B22[k, k];
                z = B22[k + 1, k];
                // determine c&s .. |c/-s s/c|T * |y/z| = |* 0|
                cs = new double[3];
                Array.Copy(Givens(y, z), cs, cs.Length);
                // accumulate UsvdT_UbdT_b
                double[] summer2 = new double[3];
                summer2[1] = +cs[1] * U[k + p - 1, 1] - cs[2] * U[k +
p, 1];
                summer2[2] = +cs[2] * U[k + p - 1, 1] + cs[1] * U[k +
p, 1];
                U[k + p - 1, 1] = summer2[1];
                U[k + p, 1] = summer2[2];
                // 'apply givins transformation to B22 B22
B22=GTTempB22
                summer = new double[3, B22.GetLength(1) - 1 + 1];
                for (i = 1; i <= B22.GetLength(1) - 1; i++)
                {
                    summer[1, i] = cs[1] * B22[k, i] - cs[2] * B22[k +
1, i];
                    summer[2, i] = cs[2] * B22[k, i] + cs[1] * B22[k +
1, i];
```

```
                }
                for (i = 1; i <= B22.GetLength(1) - 1; i++)
                {
                    B22[k, i] = summer[1, i];
                    B22[k + 1, i] = summer[2, i];
                }
                if (k < npq - 1)
                {
                    y = B22[k, k + 1];
                    z = B22[k, k + 2];
                }
            }
              }
```

Organizing the Results

Now that we have all of the necessary elements for our SVD, we need to form it. This will require multiplying out V_{bd} and V_{Σ} and U_{bd} and U_{Σ} for our explicit procedures. We also need to make the diagonal vector of singular values positive and descending. We do this with one of two organization subroutines.

For explicit systems we have:

```
        public void Organize_svd(double[,] Ubd, double[,] Usvd,
double[,] RawSigma, double[,] Vbd, double[,] Vsvd, ref double[,] U, ref
double[,] Sigma, ref double[,] V)
        {
            // Makes positive SVs descend. Forms U and V. Does not
overwrite any of the arguments.
            int m = Ubd.GetLength(0) - 1;
            int n = Vbd.GetLength(0) - 1;
            int i = 0, j = 0, k = 0;
            U = Matrix_Multiply(Ubd, Usvd); // change back to
parallel_multiply
            V = Matrix_Multiply(Vbd, Vsvd);
            double[,] localSigma = new double[m + 1, n + 1];
            Array.Copy(RawSigma, localSigma, RawSigma.Length);
            double Smax = 0;
            int xcol = 0;
            for (i = 1; i <= n; i++)
            {
                Smax = System.Math.Abs(localSigma[i, i]);
                for (j = i; j <= n; j++)
                {
                    if (System.Math.Abs(localSigma[j, j]) > Smax)
                    {
                        Smax = System.Math.Abs(localSigma[j, j]);
                        xcol = j;
                    }
                }
                if (Smax != System.Math.Abs(localSigma[i, i]))
                {
                    double tempS = 0;
                    double tempv = 0;
```

```
                    double tempU = 0;
                    tempS = localSigma[i, i];
                    localSigma[i, i] = localSigma[xcol, xcol];
                    localSigma[xcol, xcol] = tempS;
                    for (k = 1; k <= n; k++)
                    {
                        tempv = V[k, xcol];
                        V[k, xcol] = V[k, i];
                        V[k, i] = tempv;
                    }
                    for (k = 1; k <= m; k++)
                    {
                        tempU = U[k, xcol];
                        U[k, xcol] = U[k, i];
                        U[k, i] = tempU;
                    }
                }
            }
            Sigma = localSigma;
            // make all SVs positive
            for (i = 1; i <= n; i++)
            {
                if (Sigma[i, i] < 0)
                {
                    Sigma[i, i] = -Sigma[i, i];
                    for (j = 1; j <= m; j++)
                    {
                        U[j, i] = -U[j, i];
                    }
                }
            }
            if (null != done) done();
              }
```

For implicit systems we have:

```
        public void Organize_implicit_svd(double[,] UsvdT_UbdT_b,
double[,] Vsvd, double[,] Vbd, double[,] RawSigma, ref double[,] Sigma,
ref double[,] V, ref double[,] Perm_UsvdT_UbdT_b)
        {
            // Makes positive SVs ascend. Forms V. Does not overwrite
any of the arguments.
            int m = UsvdT_UbdT_b.GetLength(0) - 1;
            int n = Vbd.GetLength(0) - 1;
            Perm_UsvdT_UbdT_b = new double[m + 1, 2];
            Sigma = new double[m + 1, n + 1];
            V = new double[n + 1, n + 1];

            Array.Copy(UsvdT_UbdT_b, Perm_UsvdT_UbdT_b,
UsvdT_UbdT_b.Length);
            double[,] LocalRawSigma = new double[m + 1, n + 1];
            Array.Copy(RawSigma, LocalRawSigma, RawSigma.Length);
            int i = 0, j = 0;
            V = Matrix_Multiply(Vbd, Vsvd);
            double Smax = 0;
            int xcol = 0;
```

```
            for (i = 1; i <= n; i++)
            {
                Smax = System.Math.Abs(LocalRawSigma[i, i]);
                for (j = i; j <= n; j++)
                {
                    if (System.Math.Abs(LocalRawSigma[j, j]) > Smax)
                    {
                        Smax = System.Math.Abs(LocalRawSigma[j, j]);
                        xcol = j;
                    }
                }
                if (Smax > System.Math.Abs(LocalRawSigma[i, i]))
                {
                    double tempS = 0;
                    double tempv = 0;
                    double tempU = 0;
                    tempS = LocalRawSigma[i, i];
                    LocalRawSigma[i, i] = LocalRawSigma[xcol, xcol];
                    LocalRawSigma[xcol, xcol] = tempS;
                    for (j = 1; j <= n; j++)
                    {
                        tempv = V[j, xcol];
                        V[j, xcol] = V[j, i];
                        V[j, i] = tempv;
                    }
                    tempU = Perm_UsvdT_UbdT_b[xcol, 1];
                    Perm_UsvdT_UbdT_b[xcol, 1] = Perm_UsvdT_UbdT_b[i,
1];
                    Perm_UsvdT_UbdT_b[i, 1] = tempU;
                }
            }
            Sigma = LocalRawSigma;
            for (i = 1; i <= n; i++)
            {
                if (Sigma[i, i] < 0)
                {
                    Sigma[i, i] = -Sigma[i, i];
                    Perm_UsvdT_UbdT_b[i, 1] = -Perm_UsvdT_UbdT_b[i, 1];
                }
            }
            if (null != done) done();
              }
```

Finally, we can solve the system Ax=b with one of two methods: the pseudo-inverse technique or the summation approach for either the explicit or implicit case. The summation methods below do not provide for re-evaluation of rank, but this functionality could easily be added.

For the explicit case we have:

```
        public double[,] PI_Solve_svd(double[,] BigU, double[,] SIGMA,
double[,] BigV, double[,] b)
        { // determine x for Ax=b
            // Golub/Van Loan section 5.5.4. Does not overwrite any of
the arguments.
```

```
            int m = b.GetLength(0) - 1;
            int n = SIGMA.GetLength(1) - 1;
            int i = 0, j = 0;
            SVDRANK = n;
            double[,] Temp = new double[n + 1, n + 1]; // just the
first n rows of sigma
            double FnormS = Frobenius_norm(SIGMA);
            // form temp = pseudoinverse ie sigma plus
            for (i = 1; i <= n; i++)
            {
                if (System.Math.Abs(SIGMA[i, i]) >
SVD_Rank_Determination_Threshold * FnormS)
                {
                    Temp[i, i] = 1 / SIGMA[i, i];
                }
                else
                {
                    Temp[i, i] = 0;
                    SVDRANK = SVDRANK - 1;
                }
            }
            // make temp = V X Sigma Plus
            Temp = Matrix_Multiply(BigV, Temp);
            // make the first n rows of U transpose (a thin U)
            double[,] tUT = new double[n + 1, m + 1];
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    tUT[j, i] = BigU[i, j];
                }
            }
            // make V X Sigma Plus X the first n rows of U transpose
            double[,] temp2 = new double[n + 1, m + 1];
            temp2 = Matrix_Multiply(Temp, tUT);
            double[,] Summer = new double[n + 1, 2];
            for (i = 1; i <= n; i++)
            {
                for (j = 1; j <= m; j++)
                {
                    Summer[i, 1] = temp2[i, j] * b[j, 1] + Summer[i,
1];
                }
            }
            if (null != done) done();
            return Summer;
        }

        public double[,] SVD_Axb_by_Sum(double[,] BigU, double[,]
Sigma, double[,] BigV, double[,] b)
        {
            // Golub/Van Loan Theorem 5.5.1. Does not overwrite any of
the arguments.
            int m = b.GetLength(0) - 1;
            int n = BigV.GetLength(1) - 1;
            int i = 0, j = 0;
            double[] uiTb_over_sigma = new double[n + 1];
```

```
            double[,] Xls = new double[n + 1, n + 1];
            for (i = 1; i <= svd_Rank; i++)
            {
                double summer = 0;
                for (j = 1; j <= m; j++)
                {
                    summer = summer + BigU[j, i] * b[j, 1]; // note
that BigU(j,i)=BigU Transpose
                }
                uiTb_over_sigma[i] = summer / Sigma[i, i];
                for (j = 1; j <= n; j++)
                {
                    Xls[j, i] = uiTb_over_sigma[i] * BigV[j, i];
                }
            }
            double[,] result = new double[n + 1, 2];
            for (i = 1; i <= n; i++)
            {
                for (j = 1; j <= svd_Rank; j++)
                {
                    result[i, 1] = Xls[i, j] + result[i, 1];
                }
            }
            if (null != done) done();
            return result;
              }
```

For the implicit case we have:

```
        public double[,] PI_Solve_SVD_implicit(double[,] encoded_BD,
double[,] V, double[,] sigma, double[,] UsvdT_UbdT_b)
        {
            // Golub/Van Loan section 5.5.4
            int i = 0;
            int m = UsvdT_UbdT_b.GetLength(0) - 1;
            int n = encoded_BD.GetLength(1) - 1;
            double[,] Identity = new double[m + 1, m + 1];
            SVDRANK = n;
            double[,] sigmaplus = new double[n + 1, m + 1];
            // form pseudoinverse ie sigma plus. Also SVDRank is
performed again providing the user with the option of evaluating
            // different Machine_Error values without the need for
reprocessing the matrices from scratch
            double FnormS = Frobenius_norm(sigma);
            for (i = 1; i <= n; i++)
            {
                if (System.Math.Abs(sigma[i, i]) >
SVD_Rank_Determination_Threshold * FnormS)
                {
                    sigmaplus[i, i] = 1 / sigma[i, i];
                }
                else
                {
                    sigmaplus[i, i] = 0;
                    SVDRANK = SVDRANK - 1;
                }
            }
```

```
            double[,] BigVsigmaplus = Matrix_Multiply(V, sigmaplus);
            if (null != done) done();
            return Matrix_Multiply(BigVsigmaplus, UsvdT_UbdT_b);
        }

        public double[,] SVD_Axb_by_Sum_Implicit(double[,]
UsvdT_UbdT_b, double[,] Sigma, double[,] BigV, double[,] b)
        {
            // Golub/Van Loan Theorem 5.5.1
            int m = UsvdT_UbdT_b.GetLength(0) - 1;
            int n = Sigma.GetLength(1) - 1;
            int i = 0, j = 0;
            double[] uiTb_over_sigma = new double[n + 1];
            double[,] Xls = new double[n + 1, n + 1];
            for (i = 1; i <= svd_Rank; i++)
            {
                uiTb_over_sigma[i] = UsvdT_UbdT_b[i, 1] / Sigma[i, i];
                for (j = 1; j <= n; j++)
                {
                    Xls[j, i] = uiTb_over_sigma[i] * BigV[j, i];
                }
            }
            double[,] result = new double[n + 1, 2];
            for (i = 1; i <= n; i++)
            {
                for (j = 1; j <= svd_Rank; j++)
                {
                    result[i, 1] = Xls[i, j] + result[i, 1];
                }
            }
            if (null != done) done();
            return result;
              }
```

Making the Parts Fit Together

We have just seen the SVD problem implemented in several steps: bidiagonalization, the SVD step, organization, and solutions. For each of the steps, several variants have been presented. It may be unclear which variant of each step functions with the appropriate variant of the following step. Below are several macro functions that show these relationships. These macros take A and b as arguments and return x for the problem Ax=b. They can of course be modified to generate intermediate output such as U, $\sum$, and V.

```
        public double[,] SVD_Explicit_onestep(double[,] a, double[,] b)
        {
            double[,] BD = null;
            double[,] UBD = null;
            double[,] Vbd = null;
            Bidiagonalize_explicit_Givens(a, ref BD, ref UBD, ref Vbd);
            double[,] Usvd = null;
            double[,] raw_sigma = null;
            double[,] Vsvd = null;
            Algo862_explicit(BD, ref Usvd, ref raw_sigma, ref Vsvd);
```

```
            double[,] U = null;
            double[,] Sigma = null;
            double[,] V = null;
            Organize_svd(UBD, Usvd, raw_sigma, Vbd, Vsvd, ref U, ref
Sigma, ref V);
            return PI_Solve_svd(U, Sigma, V, b);
        }


        public double[,] SVD_Explicit_U1_Rbidiagonal_onestep(double[,]
a, double[,] b)
        {
            double[,] rbd = null;
            double[,] Urbd = null;
            double[,] Vrbd = null;
            R_Bidiagonalize_full(a, ref rbd, ref Urbd, ref Vrbd);
            double[,] Usvd = null;
            double[,] raw_sigma = null;
            double[,] Vsvd = null;
            Algo862_explicit(rbd, ref Usvd, ref raw_sigma, ref Vsvd);
            double[,] U = null;
            double[,] Sigma = null;
            double[,] V = null;
            Organize_svd(Urbd, Usvd, raw_sigma, Vrbd, Vsvd, ref U, ref
Sigma, ref V);
            return PI_Solve_svd(U, Sigma, V, b);
        }


        public double[,] SVD_Explicit_U_Rbidiagonal_onestep(double[,]
a, double[,] b)
        {
            double[,] rbd = null;
            double[,] Urbd = null;
            double[,] Vrbd = null;
            R_Bidiagonalize_full(a, ref rbd, ref Urbd, ref Vrbd);
            double[,] Usvd = null;
            double[,] raw_sigma = null;
            double[,] Vsvd = null;
            Algo862_explicit(rbd, ref Usvd, ref raw_sigma, ref Vsvd);
            double[,] U = null;
            double[,] Sigma = null;
            double[,] V = null;
            Organize_svd(Urbd, Usvd, raw_sigma, Vrbd, Vsvd, ref U, ref
Sigma, ref V);
            return PI_Solve_svd(U, Sigma, V, b);
        }


        public double[,] SVD_Implicit_onestep(double[,] a, double[,] b)
        {
            double[,] BD = null;
            double[,] Vbd = null;
            Bidiagonalize_implicit_Givens(a, ref BD, ref Vbd);
            double[,] UbdTb = null;
            UbdTb = Form_UbdTb_Givens(BD, b);
            double[,] UsvdT_UbdT_b = null;
```

```
            double[,] rawsigma = null;
            double[,] Vsvd = null;
            Algo862_implicit(BD, UbdTb, ref UsvdT_UbdT_b, ref rawsigma,
ref Vsvd);
            double[,] permuted_UsvdT_UbdT_b = null;
            double[,] Sigma = null;
            double[,] V = null;
            Organize_implicit_svd(UsvdT_UbdT_b, Vsvd, Vbd, rawsigma,
ref Sigma, ref V, ref permuted_UsvdT_UbdT_b);
            return PI_Solve_SVD_implicit(BD, V, Sigma,
permuted_UsvdT_UbdT_b);
        }


        public double[,] SVD_Implicit_Rbidiagonal_onestep(double[,] a,
double[,] b)
        {
            double[,] rbd = null;
            double[,] Vbd = null;
            double[,] UbdTb = null;
            R_Bidiagonalize_implicit(a, b, ref rbd, ref Vbd, ref
UbdTb);
            double[,] UsvdT_UbdT_b = null;
            double[,] rawsigma = null;
            double[,] Vsvd = null;
            Algo862_implicit(rbd, UbdTb, ref UsvdT_UbdT_b, ref
rawsigma, ref Vsvd);
            double[,] permuted_UsvdT_UbdT_b = null;
            double[,] Sigma = null;
            double[,] V = null;
            Organize_implicit_svd(UsvdT_UbdT_b, Vsvd, Vbd, rawsigma,
ref Sigma, ref V, ref permuted_UsvdT_UbdT_b);
            return PI_Solve_SVD_implicit(rbd, V, Sigma,
permuted_UsvdT_UbdT_b);
        }


        public double[,] SVD_Implicit_Rbidiagonal_V1_onestep(double[,]
a, double[,] b)
        {
            double[,] QR = null;
            double[,] R1BD = null;
            double[,] ur1 = null;
            double[,] vbd = null;
            R_Bidiagonalize_implicit_1(a, ref QR, ref R1BD, ref ur1,
ref vbd);
            double[,] UbdTb = null;
            UbdTb = Form_UbdTb_implicit_R_BD_1(QR, ur1, b);
            double[,] UsvdT_UbdT_b = null;
            double[,] rawsigma = null;
            double[,] Vsvd = null;
            Algo862_implicit(R1BD, UbdTb, ref UsvdT_UbdT_b, ref
rawsigma, ref Vsvd);
            double[,] permuted_UsvdT_UbdT_b = null;
            double[,] Sigma = null;
            double[,] V = null;
```

```
            Organize_implicit_svd(UsvdT_UbdT_b, Vsvd, vbd, rawsigma,
ref Sigma, ref V, ref permuted_UsvdT_UbdT_b);
            return PI_Solve_SVD_implicit(QR, V, Sigma,
permuted_UsvdT_UbdT_b);
        }


        public double[,] SVD_Implicit_Rbidiagonal_V2_onestep(double[,]
a, double[,] b)
        {
            double[,] qr = null;
            double[,] r1bd = null;
            double[,] vbd = null;
            R_Bidiagonalize_implicit_2(a, ref qr, ref r1bd, ref vbd);
            double[,] UbdTb = null;
            UbdTb = Form_UbdTb_implicit_R_BD_2(qr, r1bd, b);
            double[,] UsvdT_UbdT_b = null;
            double[,] rawsigma = null;
            double[,] Vsvd = null;
            Algo862_implicit(r1bd, UbdTb, ref UsvdT_UbdT_b, ref
rawsigma, ref Vsvd);
            double[,] permuted_UsvdT_UbdT_b = null;
            double[,] Sigma = null;
            double[,] V = null;
            Organize_implicit_svd(UsvdT_UbdT_b, Vsvd, vbd, rawsigma,
ref Sigma, ref V, ref permuted_UsvdT_UbdT_b);
            return PI_Solve_SVD_implicit(qr, V, Sigma,
permuted_UsvdT_UbdT_b);
              }
```

The Underdetermined Case

Our Golub-Reinsch SVD strategy enables us to solve an underdetermined system. In an underdetermined system Ax=b, we have A^{mxn} where $m < n$. Underdetermined systems either have an infinity of solutions or no solution at all. The solution the SVD provides is a minimum norm solution. I have not extended this functionality to the Chan R-Bidiagonalization strategy. Chan R-Bidiagonalization is employed when m>>n.

Examples

We do not have room for a detailed example for all of the implementation sequences given above. We will first look at the fully explicit Golub-Reinsch SVD procedure.

$$A = \begin{bmatrix} 8.2 & 16.4 & 2.1 & 10.3 \\ 9.4 & 18.8 & 5.2 & 14.6 \\ 11.1 & 22.2 & 7.5 & 18.6 \\ 14.7 & 29.4 & 10.4 & 25.1 \\ 6.2 & 12.4 & 3.3 & 9.5 \\ 2.9 & 5.8 & 4.6 & 7.5 \end{bmatrix}, \; b = \begin{pmatrix} 88.5 & 121 & 152.4 & 205.1 & 78.9 & 58.3 \end{pmatrix}^T$$

$$BD = \begin{bmatrix} -23.27552 & -61.62521 & 0 & 0 \\ 0 & -3.735111 & 5.001186 & 0 \\ 0 & 0 & -7.497559E\text{-}15 & 8.892092E\text{-}16 \\ 0 & 0 & 0 & 1.131181E\text{-}15 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$U_{bd} = \begin{bmatrix} -0.3523 & 0.676562 & -0.10117 & -0.55631 & 0.313738 & 0 \\ -0.40386 & 0.142852 & 0.000942 & 0.67927 & 0.443215 & 0.398301 \\ -0.4769 & -0.13934 & 0.747542 & -0.18486 & -0.32176 & 0.23797 \\ -0.63156 & -0.29047 & -0.22093 & 0.085481 & -0.00248 & -0.6787 \\ -0.26637 & 0.123626 & -0.55204 & 0.036094 & -0.67972 & 0.38169 \\ -0.12459 & -0.63465 & -0.27819 & -0.43166 & 0.373591 & 0.422382 \end{bmatrix}$$

$$V_{bd} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.75539 & 0.564159 & 0.333333 \\ 0 & 0.234272 & -0.70758 & 0.666667 \\ 0 & 0.611967 & -0.4255 & -0.66667 \end{bmatrix}$$

$$\Sigma_{unorganized} = diag(65.96743, \text{-}5.165079, 2.177718E\text{-}15, 9.93681E\text{-}16)$$

$$U_{\Sigma} = \begin{bmatrix} 0.998579 & -0.0533 & -7.14E\text{-}17 & 2.07E\text{-}17 & 0.000000 & 0.000000 \\ 0.053296 & 0.998579 & 1.35E\text{-}15 & -3.91E\text{-}16 & 0.000000 & 0.000000 \\ -4.59E\text{-}19 & -1.40E\text{-}15 & 0.960309 & -0.27894 & 0.000000 & 0.000000 \\ -1.06E\text{-}52 & -5.29E\text{-}47 & 0.278939 & 0.960309 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{bmatrix}$$

$$V_{\Sigma} = \begin{bmatrix} -0.352332108 & -0.240167166 & 0.763045782 & -0.485739593 \\ -0.935866315 & 0.086242956 & -0.288198431 & 0.183461322 \\ 0.004040496 & -0.966892902 & -0.215239586 & 0.13701719 \\ -6.19\text{E-}36 & 2.42\text{E-}31 & 0.537005325 & 0.843578853 \end{bmatrix}$$

$$\sum = diag(65.96743,\ 5.165079,\ 2.177718\text{E-}15,\ 9.93681\text{E-}16)$$

$$U = \begin{bmatrix} -0.31574297 & -0.694376171 & -0.252329588 & -0.506010354 & 0.31373778 & 0 \\ -0.395670384 & -0.164172471 & 0.190380111 & 0.652046561 & 0.443214834 & 0.398300946 \\ -0.483644396 & 0.113728434 & 0.666306436 & -0.386041939 & -0.321763545 & 0.237969548 \\ -0.646147847 & 0.256396353 & -0.188312426 & 0.143713132 & -0.00247848 & -0.678699255 \\ -0.259406924 & -0.137647182 & -0.520062089 & 0.188647538 & -0.679721475 & 0.381690353 \\ -0.158241585 & 0.627111733 & -0.387549669 & -0.336925954 & 0.373590678 & 0.422382087 \end{bmatrix}$$

$$V = \begin{bmatrix} -0.352332108 & -0.240167166 & 0.763045782 & -0.485739593 \\ -0.704664215 & -0.480334333 & -0.160129684 & 0.497077214 \\ -0.222106135 & 0.704360284 & 0.442786415 & 0.508414836 \\ -0.574438243 & 0.464193118 & -0.442786415 & -0.508414836 \end{bmatrix}$$

$$x_{ls} = (1, 2, 3, 4)^T$$

Next we will look at the fully explicit Chan R-Bidiagonalization SVD procedure using the same A and b as above.

$$RBD = \begin{bmatrix} 23.27552 & 61.62521 & 0.00000 & 0.00000 \\ 0.00000 & 3.73511 & -5.00119 & 0.00000 \\ 0.00000 & 0.00000 & 1.85\text{E-}14 & -2.38\text{E-}15 \\ 0.00000 & 0.00000 & 0.00000 & 4.81\text{E-}16 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 \end{bmatrix}$$

$$U_{RBD} = \begin{bmatrix} 0.352301419 & -0.676561627 & -0.461293525 & -0.144182938 & 0.33410149 & -0.27008116 \\ 0.403857724 & -0.142851661 & -0.084697494 & 0.808973687 & -0.365974697 & 0.144716176 \\ 0.476895823 & 0.139342912 & 0.643661676 & 0.081252452 & 0.310473945 & -0.485651667 \\ 0.631564739 & 0.290468779 & -0.174604972 & -0.509618395 & -0.467816497 & 0.087771232 \\ 0.266374244 & -0.12362631 & 0.24542344 & -0.079757096 & 0.430745752 & 0.813403673 \\ 0.124594404 & 0.634654035 & -0.524418443 & 0.228251595 & 0.503637238 & -0.030438936 \end{bmatrix}$$

$$V_{RBD} = \begin{bmatrix} 1.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.75539 & 0.56416 & 0.33333 \\ 0.00000 & 0.23427 & -0.70758 & 0.66667 \\ 0.00000 & 0.61197 & -0.42550 & -0.66667 \end{bmatrix}$$

$$\sum_{unorganized} = diag(-65.96743,\ 5.165079,\ -5.298976\text{E-}15,\ 4.293314\text{E-}16)$$

$$U_{\Sigma} = \begin{bmatrix} 0.99858 & -0.05330 & 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.05330 & 0.99858 & 0.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.99915 & 0.04112 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & -0.04112 & 0.99915 & 0.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 & 1.00000 & 0.00000 \\ 0.00000 & 0.00000 & 0.00000 & 0.00000 & 0.00000 & 1.00000 \end{bmatrix}$$

$$V_{\Sigma} = \begin{bmatrix} -0.352332108 & -0.240167166 & 0.806451341 & -0.409656018 \\ -0.935866315 & 0.086242956 & -0.304592485 & 0.154724951 \\ 0.004040496 & -0.966892902 & -0.227483405 & 0.115555572 \\ -7.87\text{E-}33 & 1.60\text{E-}30 & 0.452891768 & 0.891565503 \end{bmatrix}$$

$$\sum = diag(65.96743,\ 5.165079,\ 2.177718\text{E-}15,\ 9.93681\text{E-}16)$$

$$U = \begin{bmatrix} -0.315742979 & -0.694376192 & 0.454974239 & -0.163030298 & 0.33410149 & -0.27008116 \\ -0.395670392 & -0.164172468 & 0.117892481 & 0.80480647 & -0.365974697 & 0.144716176 \\ -0.483644411 & 0.113728437 & -0.639775959 & 0.10765239 & 0.310473945 & -0.485651667 \\ -0.646147848 & 0.256396352 & 0.153500743 & -0.516367434 & -0.467816497 & 0.087771232 \\ -0.259406932 & -0.137647179 & -0.248495617 & -0.069597325 & 0.430745752 & 0.813403673 \\ -0.158241581 & 0.627111722 & 0.53336102 & 0.20649338 & 0.503637238 & -0.030438936 \end{bmatrix}$$

$$V = \begin{bmatrix} -0.352332108 & -0.240167166 & 0.806451341 & -0.409656018 \\ -0.704664215 & -0.480334333 & -0.207458896 & 0.479257842 \\ -0.222106135 & 0.704360284 & 0.39153355 & 0.548859667 \\ -0.574438243 & 0.464193118 & -0.39153355 & -0.548859667 \end{bmatrix}$$

$$x_{ls} = (1, 2, 3, 4)^T$$

Finally, let us look at the implicit Chan R-bidiagonalization via the R_Bidiagonalize_implicit_2 procedure.

$$Q\,\mathrm{R}_{\text{encoded}} = \begin{bmatrix} -23.27552 & -46.55105 & -14.43705 & -37.71258 \\ -5.67696 & 0.00000 & -0.31973 & -0.31973 \\ 4.63473 & 1.00000 & -4.40218 & -4.40218 \\ -3.54067 & 1.00000 & 0.34447 & 0.00000 \\ -0.21106 & 0.22361 & -4.90490 & 0.25220 \\ -0.21184 & 0.00000 & 87.96641 & 0.25856 \end{bmatrix}$$

$$R1BD_{encoded} = \begin{bmatrix} -23.275523 & -61.625214 & 0 & 0 \\ 0 & 3.7351114 & -5.00118631 & 0 \\ 0 & -27.609018 & -7.51E-15 & 8.50E-16 \\ 0 & 1.27E-16 & 0.0404787 & -9.63E-16 \end{bmatrix}$$

$$V_{RBD} = \begin{bmatrix} 1.00000 & 0.00000 & 0.00000 & 0.00000 \\ 0.00000 & 0.75539 & 0.56416 & 0.33333 \\ 0.00000 & 0.23427 & -0.70758 & 0.66667 \\ 0.00000 & 0.61197 & -0.42550 & -0.66667 \end{bmatrix}$$

$$U_{BD}^{T} b = (-310.539,\ 30.89647,\ 6.61E\text{-}14,\ -1.83E\text{-}14)^{T}$$

$$V_{\Sigma} = \begin{bmatrix} -0.352332108 & -0.240167166 & 0.790350836 & -0.439917462 \\ -0.935866315 & 0.086242956 & -0.298511408 & 0.166154542 \\ 0.004040496 & -0.966892902 & -0.222941782 & 0.124091705 \\ -5.93E\text{-}36 & 2.31E\text{-}31 & 0.486347053 & 0.873765726 \end{bmatrix}$$

$$\Sigma_{unorganized} = diag(65.96743,\ 5.165079,\ 2.14E\text{-}15,\ -8.62E\text{-}16)$$

$$U^{T} b = U_{\Sigma}^{T} U_{RBD}^{T} b = (-311.744,\ 14.30219,\ 4.87E\text{-}14,\ -7.81E\text{-}15)^{T}$$

$$\Sigma = diag(65.96743,\ 5.165079,\ 2.14E\text{-}15,\ 8.62E\text{-}16)$$

$$V = \begin{bmatrix} -0.352332108 & -0.240167166 & 0.790350836 & -0.439917462 \\ -0.704664215 & -0.480334333 & -0.189151354 & 0.486774114 \\ -0.222106135 & 0.704360284 & 0.412048129 & 0.533630766 \\ -0.574438243 & 0.464193118 & -0.412048129 & -0.533630766 \end{bmatrix}$$

$$U^{T} b_{organized} = (-311.744,\ 14.30219,\ 4.87E\text{-}14,\ -7.81E\text{-}15)^{T}$$

$$x_{ls} = (1, 2, 3, 4)^{T}$$

Computational Overhead and Memory Demand

In Chapter 2 we considered a thoroughly populated (no zero elements), 200 × 200 element symmetric positive definite matrix in some timing experiments. The following table shows the performance for certain of the SVD procedures.

Procedure	Time, ms
Explicit Bidiagonalization	24,500
Implicit Bidiagonalization	16,500
Implicit R-Bidiagonalization (Variant 2)	17,860

Here are some results for a system where A is 2500 × 2.

Procedure	Time, ms
Explicit Bidiagonalization	289,700
Explicit R-Bidiagonalization	259,000
Explicit R1 Bidiagonalization	280
Implicit Bidiagonalization	233
Implicit R-Bidiagonalization (Variant 2)	16

Memory limitations and not processing power constrain applications with matrix computations on personal computers. There are many things we can do to improve memory use. The implementations provided in this book sacrifice memory in favor of readability. The bidiagonal matrices and the Sigma matrices can be stored as vectors. Intermediate matrices we form along the path to a solution can be disposed of immediately after they serve their purpose. The interface I use can be greatly simplified. The list goes on.

6. Sensitivity of Linear Systems, Algorithmic Stability and Error Analysis for Least Squares Solutions

Sensitivity of Linear Systems

A More Detailed Look at Norms

We have casually made reference to norms throughout the preceding chapters without discussing what they are. We have defined the Euclidian norm for vectors, and the Frobenius norm for matrices, but it turns out there are other important norms. A better understanding of these functions that yield scalar quantities is important as we explore the subject of error analysis. The following is a brief outline of critical details. More complete and rigorous treatments are provided in Golub and Watkins.

The absolute value of a scalar is the most fundamental norm. It represents the magnitude of the scalar. The absolute value is the length of a line from zero to the scalar's value on the real number line. Vector norms extend this concept to a multi-dimensional vector space. One type of vector norm is the *p-norm*: $\|x\|_p = \left(\sum_{i=1}^{n} x_i^p \right)^{1/p}$. The important p-norms are the 1, 2, and infinity norms:

$\|x\|_1 = \sum_{i=1}^{n} |x_i|$, $\|x\|_2 = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}$ and $|x|_\infty = \max |x_i|$. The 2 norm is the Euclidian norm we have previously discussed.

Here are some functions for determining 1, 2 and infinity vector norms:

```
public double Vector_One_Norm(double[,] vector)
{
    int i = 0;
    double result = 0;
    int m = vector.GetLength(0) - 1;
    for (i = 1; i <= m; i++)
    {
        result = System.Math.Abs(vector[i, 1]) + result;
    }
    return result;
}

public double Vector_Two_Norm(double[,] vector)
{
    int i = 0;
    double result = 0;
    int m = vector.GetLength(0) - 1;
    for (i = 1; i <= m; i++)
    {
```

```
            result = Math.Pow(vector[i, 1], 2) + result;
        }
        return Math.Pow(result, 0.5);
    }

    public double Vector_Infinity_norm(double[,] vector)
    {
        int i = 0;
        double result = 0;
        int m = vector.GetLength(0) - 1;
        double[] temp = new double[m + 1];
        for (i = 1; i <= m; i++)
        {
            temp[i] = System.Math.Abs(vector[i, 1]);
        }
        result = temp[1];
        for (i = 2; i <= m; i++)
        {
            if (temp[i] > result)
            {
                result = temp[i];
            }
        }
        return result;
      }
```

Matrix norms continue the sequence from absolute values for scalars to vector norms to arrive at a means of quantifying the distance on the space of matrices. We have already discussed the Frobenius norm: $\|A\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \right)^{1/2}$. There are also matrix p norms, again the most important of which are the 1, 2, and infinity matrix norms. The matrix 1 norm is the maximum column sum of absolute values for the matrix. The infinity norm is the maximum row sum of absolute values for the matrix. Finally, the matrix 2 norm is the largest singular value for the matrix. The Frobenius norm should not be confused with the matrix 2 norm. The 1 and infinity norms are easy to compute. The matrix 2 norm requires an SVD computation; however, a rough estimate can be obtained from the following inequalities:

$$\frac{1}{\sqrt{n}}\|A\|_\infty \le \|A\|_2 \le \sqrt{m}\,\|A\|_\infty$$

$$\frac{1}{\sqrt{m}}\|A\|_1 \le \|A\|_2 \le \sqrt{n}\,\|A\|_1$$

$$\|A\|_2 \le \sqrt{\|A\|_1 \|A\|_\infty}$$

Here are some functions for determining matrix 1, infinity, and F norms:

```
    public double Frobenius_norm(double[,] Source_Matrix)
    {
```

```
            int i = 0, j = 0;
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            double result = 0;
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    result = result + (Math.Pow(Source_Matrix[i, j],
2));
                }
            }
            result = Math.Pow(result, 0.5);
            return result;
        }

        public double Matrix_1_norm(double[,] Source_Matrix)
        {
            int i = 0, j = 0;
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            double[,] ColSum = new double[n + 1, 2];
            for (j = 1; j <= n; j++)
            {
                for (i = 1; i <= m; i++)
                {
                    ColSum[j, 1] = ColSum[j, 1] +
System.Math.Abs(Source_Matrix[i, j]);
                }
            }
            double result = Vector_Infinity_norm(ColSum);
            return result;
        }

        public double Matrix_infinity_norm(double[,] Source_Matrix)
        {
            int i = 0, j = 0;
            int m = Source_Matrix.GetLength(0) - 1;
            int n = Source_Matrix.GetLength(1) - 1;
            double[,] RowSum = new double[m + 1, 2];
            for (i = 1; i <= m; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    RowSum[i, 1] = RowSum[i, 1] +
System.Math.Abs(Source_Matrix[i, j]);
                }
            }
            double result = Vector_Infinity_norm(RowSum);
            return result;
              }
```

There are times when an estimate of the matrix 2 norm will not suffice and an explicit calculation of the largest singular value is required. Our procedures for the SVD involve not only the determination of the singular values, but computation of U and V as

well. The following three procedures are a Sigma only truncation of the bidiagonalization-SVD methods presented in the previous chapter.

```
        public void Bidiagonalize_bd_SigmaOnly(double[,] Source_Matrix,
ref double[,] BD)
        {
            int i = 0, j = 0, k = 0, m = 0, n = 0;
            if (Source_Matrix.GetLength(1) - 1 >
Source_Matrix.GetLength(0) - 1)
            {
                m = Source_Matrix.GetLength(1) - 1; // allow for
underdetermined system
            }
            else
            {
                m = Source_Matrix.GetLength(0) - 1;
            }
            n = Source_Matrix.GetLength(1) - 1;
            BD = new double[m + 1, n + 1];
            Array.Copy(Source_Matrix, BD, Source_Matrix.Length);
            for (j = 1; j <= n; j++)
            {
                for (i = m; i >= j + 1; i += -1)
                {
                    if (System.Math.Abs(BD[i, j]) > Givens_Zero_Value)
                    { //  If it's already zero we won't zero it.
                        double[,] temp_R = new double[3, n - j + 1 +
1];
                        double[,] temp_U = new double[m + 1, 3];
                        double[] cs = new double[3];
                        cs = Givens(BD[i - 1, j], BD[i, j]);
                        // accumulate R = BD
                        for (k = j; k <= n; k++)
                        {
                            temp_R[1, -j + 1 + k] = cs[1] * BD[i - 1,
k] - cs[2] * BD[i, k];
                            temp_R[2, -j + 1 + k] = cs[2] * BD[i - 1,
k] + cs[1] * BD[i, k];
                        }
                        for (k = 1; k <= n - j + 1; k++)
                        {
                            BD[i - 1, j + k - 1] = temp_R[1, k];
                            BD[i, j + k - 1] = temp_R[2, k];
                        }
                    }
                }
                if (j <= n - 2)
                {
                    for (i = n; i >= j + 2; i += -1)
                    {
                        double[,] temp_R = new double[m - j + 1 + 1,
3];
                        double[,] temp_V = new double[n + 1, 3];
                        double[] cs = new double[3];
                        cs = Givens(BD[j, i - 1], BD[j, i]);
                        for (k = j; k <= m; k++)
```

```
                        {
                            temp_R[-j + 1 + k, 1] = cs[1] * BD[k, i -
1] - cs[2] * BD[k, i];
                            temp_R[-j + 1 + k, 2] = cs[2] * BD[k, i -
1] + cs[1] * BD[k, i];
                        }
                        for (k = 1; k <= m - j + 1; k++)
                        {
                            BD[j + k - 1, i - 1] = temp_R[k, 1];
                            BD[j + k - 1, i] = temp_R[k, 2];
                        }
                    }
                }
            }
            if (null != done) done();
            for (i = 1; i <= n; i++)
            {
                for (k = i + 1; k <= m; k++)
                {
                    BD[k, i] = 0;
                }
            }
            for (k = 1; k <= n - 2; k++)
            {
                for (i = k + 2; i <= n; i++)
                {
                    BD[k, i] = 0;
                }
            }
        }

        public void Algo862_sigma_SigmaOnly(double[,]
bidiagonalized_sourse_array, ref double[,] Sigma)
        {
            // Golub/Van Loan 8.6.2. Does not overwrite Source_matrix
            int m = bidiagonalized_sourse_array.GetLength(0) - 1;
            int n = bidiagonalized_sourse_array.GetLength(1) - 1;
            int q = 0, p = 0, h = 0, i = 0, j = 0, k = 0;
            Sigma = new double[n + 1, n + 1];
            for (i = 1; i <= n; i++)
            {
                j = i; // copy BD to SVD
                while (j < n)
                {
                    Sigma[i, j] = bidiagonalized_sourse_array[i, j];
                    Sigma[i, j + 1] = bidiagonalized_sourse_array[i, j
+ 1];
                    j = j + 1;
                }
                Sigma[i, j] = bidiagonalized_sourse_array[i, j];
            }
            while (q < n)
            {
                // Set bi,i+1 to zero...
                for (i = 1; i <= n - 1; i++)
                {
```

```
                    if (System.Math.Abs(Sigma[i, i + 1]) <=
Machine_Error * (System.Math.Abs(Sigma[i, i]) + System.Math.Abs(Sigma[i
+ 1, i + 1])))
                    {
                        Sigma[i, i + 1] = 0;
                    }
                }
                // find the largest Q
                q = 0;
                for (i = n; i >= 1; i += -1)
                {
                    if (Sigma[i - 1, i] == 0)
                    {
                        q = q + 1;
                    }
                    else
                    {
                        break;
                    }
                }
                // and smallest P....
                p = n - q;
                for (i = n - q; i >= 2; i += -1)
                {
                    if (Sigma[i - 1, i] != 0)
                    {
                        p = p - 1;
                    }
                    else
                    {
                        break;
                    }
                }
                if (q < n)
                {
                    // if any diag on b22 is zero then zero the super
                    bool FoundZero = false;
                    for (h = p; h <= n - q; h++)
                    {
                        for (i = p; i <= n - q; i++)
                        {
                            if (Sigma[i, i] == 0)
                            {
                                Sigma[i - 1, i] = 0;
                                FoundZero = true;
                            }
                        }
                        if (FoundZero == true)
                        {
                            goto L1; // Here we use the unthinkable
goto statement. I think the code is clearer with it.
                        }
                        else
                        {
                            // apply algo 861 to B22
                            double[,] b22 = new double[n - p - q + 1 +
1, n - p - q + 1 + 1]; // form it
```

```
                            for (j = 1; j <= n - p - q + 1; j++)
                            {
                                for (k = 1; k <= n - p - q + 1; k++)
                                {
                                    b22[j, k] = Sigma[j + p - 1, k + p
- 1];
                                }
                            }
                            Algo861_sigma_SigmaOnly(b22, n, p, q);
                            for (j = 1; j <= n - p - q + 1; j++)
                            { // sub it back in

                                for (k = 1; k <= n - p - q + 1; k++)
                                {
                                    Sigma[j + p - 1, k + p - 1] =
b22[j, k];
                                }
                            }
                        }
                    }
                }
            L1: { }
            }
            if (null != done) done();
        }

        private void Algo861_sigma_SigmaOnly(double[,] B22, int n, int
p, int q)
        {
            // Golub/Van Loan 8.6.1
            int i = 0, k = 0, npq = 0;
            npq = n - p - q + 1;
            // create certain elements of T = B22transpose * b22
implicitly
            double[,] TTT = new double[3, 3];
            TTT[1, 1] = (Math.Pow(B22[npq - 1, npq - 1], 2));
            TTT[1, 2] = B22[npq - 1, npq - 1] * B22[npq - 1, npq];
            TTT[2, 1] = TTT[1, 2];
            TTT[2, 2] = (Math.Pow(B22[npq, npq], 2)) +
(Math.Pow(B22[npq - 1, npq], 2));
            // create u = eigenvalue of trailing 2x2 submatrix of T
that is closer to tnn
            double d = (TTT[1, 1] - TTT[2, 2]) / 2;
            double u1 = TTT[2, 2] - (Math.Pow(TTT[2, 1], 2)) / (d +
(System.Math.Sign(d) * (Math.Pow(((Math.Pow(d, 2)) + (Math.Pow(TTT[2,
1], 2))), 0.5))));
            double y = (Math.Pow(B22[1, 1], 2)) - u1;
            double z = B22[1, 1] * B22[1, 2];
            for (k = 1; k <= npq - 1; k++)
            {
                // find c and s such that |y z||s/-s s/c|=|* 0|
                double[] cs = new double[4];
                Array.Copy(Givens(y, z), cs, cs.Length);
                // apply givins transformation to B22 B22=TempB22G
                double[,] summer = new double[B22.GetLength(0) - 1 + 1,
3];
                for (i = 1; i <= B22.GetLength(0) - 1; i++)
```

```
                {
                    summer[i, 1] = cs[1] * B22[i, k] - cs[2] * B22[i, k
+ 1];
                    summer[i, 2] = cs[2] * B22[i, k] + cs[1] * B22[i, k
+ 1];
                }
                for (i = 1; i <= B22.GetLength(0) - 1; i++)
                {
                    B22[i, k] = summer[i, 1];
                    B22[i, k + 1] = summer[i, 2];
                }
                y = B22[k, k];
                z = B22[k + 1, k];
                // determine c&s .. |c/-s s/c|T * |y/z| = |* 0|
                cs = new double[3];
                Array.Copy(Givens(y, z), cs, cs.Length);
                // accumulate U TempU=TempUG
                // 'apply givins transformation to B22 B22
B22=GTTempB22
                summer = new double[3, B22.GetLength(1) - 1 + 1];
                for (i = 1; i <= B22.GetLength(1) - 1; i++)
                {
                    summer[1, i] = cs[1] * B22[k, i] - cs[2] * B22[k +
1, i];
                    summer[2, i] = cs[2] * B22[k, i] + cs[1] * B22[k +
1, i];
                }
                for (i = 1; i <= B22.GetLength(1) - 1; i++)
                {
                    B22[k, i] = summer[1, i];
                    B22[k + 1, i] = summer[2, i];
                }
                if (k < npq - 1)
                {
                    y = B22[k, k + 1];
                    z = B22[k, k + 2];
                }
            }
          }
```

We can use these methods to determine the matrix 2 norm.

```
public double Matrix_2_norm(double[,] A)
{
    int i = 0;
    int m = A.GetLength(0) - 1;
    int n = A.GetLength(1) - 1;
    double[,] bd = new double[m + 1, n + 1];
    Array.Copy(A, bd, A.Length);
    Bidiagonalize_bd_SigmaOnly(A, ref bd);
    double[,] sigma = null;
    Algo862_sigma_SigmaOnly(bd, ref sigma);
    double result = System.Math.Abs(sigma[1, 1]);
    for (i = 2; i <= sigma.GetLength(1) - 1; i++)
    {
        if (System.Math.Abs(sigma[i, i]) > result)
        {
```

```
                result = System.Math.Abs(sigma[i, i]);
            }
        }
        return result;
    }
```

The Condition Number and Nearness to Singularity

One might think that because a singular matrix has a determinant of zero, the determinant can be used as a measure of nearness to singularity. However, this is not the case. Nearness to singularity is best measured with the condition number. Condition numbers are defined in terms of the norms used to calculate them. For square systems we can use a condition number defined by $\kappa(A) = \|A\|\|A^{-1}\|$. For rectangular systems we can use the 2-norm condition number. The 2-norm condition number of matrix A, denoted $\kappa_2(A)$, is the ratio of the highest to lowest singular values.

```
    public double Matrix_Condition(double[,] A)
    {
        int i = 0;
        int m = A.GetLength(0) - 1;
        int n = A.GetLength(1) - 1;
        double[,] bd = new double[m + 1, n + 1];
        Array.Copy(A, bd, A.Length);
        Bidiagonalize_bd_SigmaOnly(A, ref bd);
        double[,] sigma = null;
        Algo862_sigma_SigmaOnly(bd, ref sigma);
        double result1 = System.Math.Abs(sigma[1, 1]);
        for (i = 2; i <= sigma.GetLength(1) - 1; i++)
        {
            if (System.Math.Abs(sigma[i, i]) > result1)
            {
                result1 = System.Math.Abs(sigma[i, i]);
            }
        }
        double result2 = System.Math.Abs(sigma[1, 1]);
        for (i = 2; i <= sigma.GetLength(1) - 1; i++)
        {
            if (System.Math.Abs(sigma[i, i]) < result2)
            {
                result2 = System.Math.Abs(sigma[i, i]);
            }
        }
        return result1 / result2;
    }
```

A singular matrix has a 2-norm condition of infinity. The larger the condition number is, the closer A is to being singular. An orthogonal matrix — one with maximally independent column (row) vectors — has a 2-norm condition of unity.

Sensitivity of Determined Linear Systems to Perturbations in b and A (Conditioning Error)

For most real world determined linear systems, we apply solution methods using coefficient matrices and observation vectors that have some inherent error. The sets of values we use for A and b are themselves estimates of some true sets. How will a variation in those values impact the final result x? When we vary the values of A and/or b we call that variation a *perturbation*. One form of perturbation is error. Understand that perturbation theory has applications beyond error analysis. We will not derive or discuss rigorously any of the results provided here (see Watkins, Chapter 2, for details).

Let us first consider the sensitivity of a determined linear system to perturbations in the vector b. For the system Ax=b we will add a perturbation vector ∂b, giving us a new system $A\hat{x} = b + \partial b$. We want to evaluate $x - \hat{x}$ relative to x; to do that we need the ratio of the magnitude of this quantity in vector space (a vector norm) vs. the magnitude of x in vector space (the same type of norm). Using generic norms, the formula for this relative error is $\frac{\|\partial x\|}{\|x\|} \le \kappa\ (A) \frac{\|\partial b\|}{\|b\|}$, where $\partial x = x - \hat{x}$. Similarly, we consider perturbations of the coefficient matrix A. An error perturbation ∂A is added to A, giving us $(A + \partial A)\hat{x} = b$. The formula for this relative error is $\frac{\|\partial x\|}{\|x\|} \le \kappa\ (A) \frac{\|\partial A\|}{\|A\|}$. The combined effect of perturbations in A and b is given by the following inequality:

$$\frac{\|\partial x\|}{\|x\|} \le \kappa\ (A) \left(\frac{\|\partial A\|}{\|A\|} + \frac{\|\partial b\|}{\|b\|} + \frac{\|\partial A\|}{\|A\|} \frac{\|\partial b\|}{\|b\|} \right)$$

Clearly a minor change in A or b will drastically affect x for a nearly singular A matrix. In fact, the equations above are not defined for singular matrices. Conversely, a well conditioned A matrix will produce a relatively fault-tolerant set of linear systems. It should be noted that the errors described here, frequently referred to as conditioning errors, are maximum expected errors.

We can write a small routine that calculates the relative error in x for a given predicted error in the observation vector b. The following sub takes A and b, as well as a uniform relative predicted error in the elements of b (expressed in parts per million), as arguments and returns a relative maximum condition error in x (also in parts per million).

```
        public double Square_Xerr_peterb_B(double[,] A, double[,] b,
double ppm)
        {
            double[,] db = null;
            int m = A.GetLength(0) - 1;
            double scalar = ppm * 0.000001;
            db = Scalar_multiply(b, scalar);
            double normb = 0, normdb = 0, condA = 0;
            normb = Vector_Infinity_norm(b);
            normdb = Vector_Infinity_norm(db);
```

```
            condA = Matrix_infinity_norm(A) *
Matrix_infinity_norm(Inverse_by_LU_encoded(A));
            return (condA * normdb / normb) / 0.000001;
        }
```

It makes some sense to use such a function to evaluate the effect of perturbations in b, because the elements of the observation vector are frequently obtained from a unique measurement device. Measurements within a given device's dynamic range would have similar relative error. Discovering the effect of perturbations in the coefficient matrix is a different matter. Here each column would be expected to have its own unique error profile. So, analysis of the associated perturbation effects is more application-specific.

There are linear systems where poor conditioning can be resolved through scaling. Recall that we express a linear system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{12}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{22}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{32}x_3 &= b_3 \end{aligned} \text{ in matrix form with } \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Any one of the rows in the linear system can be multiplied by a scalar. This equates to multiplying the corresponding rows in A and b by that scalar. If the coefficients of a given row are disproportionately low or high with respect to the other rows then the A matrix may be ill conditioned, but this might be corrected by scaling the row. (It should be noted here that Golub/Van Loan provides some considerably more sophisticated scaling ideas for Gaussian elimination.) There is a problem with scaling disproportionately low rows. These low numerical values are often measurements, and comparatively low measured values frequently have high relative errors that are magnified by scaling.

Whenever determined linear systems are being solved, it makes sense to evaluate the condition number to become aware of potential condition errors. Explicit calculation of relative error due to condition error as explained above provides a tool for determining whether solutions can be expected to meet application-specific quality objectives.

The Residual for a Linear System

Considering the system Ax=b, let us denote A as the matrix of coefficients and b as the observation vector. Depending on the nature of the problem we use a given technique to obtain a solution for x. Because there are always errors, the solution is always an approximation to x, so we denote our solution $\hat{x}$. We define the residual, $\hat{r}$, as $\hat{r} = b - A\hat{x}$. Clearly, for a perfect system, $\hat{r} = 0$. The variable $\hat{r}$ represents a vector. The magnitude of the residual vector (its norm) is more useful in understanding the difference between the predicted and actual values for b. The following procedure will return the 2-norm of the residual:

```
        public double Two_norm_residual(double[,] A, double[,] b,
double[,] xHat)
```

```
{
    int i = 0;
    int m = A.GetLength(0) - 1;
    double[,] rHat = new double[m + 1, 2];
    double[,] AxHat = new double[m + 1, 2];
    AxHat = Matrix_Multiply(A, xHat);
    for (i = 1; i <= m; i++)
    {
        rHat[i, 1] = b[i, 1] - AxHat[i, 1];
    }
    return Vector_Two_Norm(rHat);
}
```

The magnitude of the residual relative to the magnitude of b is particularly informative. The statistic $\frac{\|\hat{r}\|}{\|b\|}$ provides an overall measure of the goodness of fit for the result $\hat{x}$. The following function takes A, b, and $\hat{x}$ as arguments and returns a 2-norm relative residual statistic:

```
public double Relative_residual(double[,] A, double[,] b,
double[,] xHat)
{
    return Two_norm_residual(A, b, xHat) / Vector_Two_Norm(b);
}
```

Sensitivity of Least Squares Solutions for Full Rank Linear Systems to Perturbations in b and A

As detailed in Watkins, the condition error for least squares solutions where the A matrix has full rank is given in terms of the perturbation in b ($\|\partial b\|_2 / \|b\|_2$) and A ($\|\partial A\|_2 / \|A\|_2$) by:

$$\frac{\|\partial x\|_2}{\|x\|_2} \le 2\kappa_2(A)\frac{\|b\|_2}{\|Ax\|_2}\frac{\|\partial b\|_2}{\|b\|_2} + 2\kappa_2(A)^2\frac{\|Ax-b\|_2}{\|Ax\|_2}\frac{\|\partial A\|_2}{\|A\|_2} + 2\kappa_2(A)\frac{\|\partial A\|_2}{\|A\|_2}$$

Note that for a perturbation in A, the condition error is a function of $\kappa_2(A)^2$, so a moderately ill conditioned A matrix can yield a substantial condition error.

The equation above can be used to develop a function that returns the maximum conditioning error given a uniform perturbation in the b vector. The function will be essentially the same as that for the determined system, and its encoding is left as an exercise. Generalization of this procedure to encompass rank deficient problems is discussed in Bjorck.

The Nearness of $\hat{x}$ to the True x

For any linear system with a solution $\hat{x}$ there is a theoretical true x. Agreement between $\hat{x}$ and x is called accuracy and is expressed in terms of the error given by $\left\|x-\hat{x}\right\|$. The lower this norm is, the higher the accuracy. The relative error is given by $\frac{\left\|x-\hat{x}\right\|}{\left\|x\right\|}$. For determined systems the formula for the maximum relative error is $\frac{\left\|x-\hat{x}\right\|}{\left\|x\right\|} \le \kappa(A)\frac{\left\|\hat{r}\right\|}{\left\|b\right\|}$, where $\hat{r}$ is the residual. For least squares solutions obtained using the normal equations method, the maximum relative error is $\frac{\left\|x-\hat{x}\right\|_2}{\left\|x\right\|_2} \le \kappa_2(A)^2 u$, where u is machine round off error. To obtain the maximum error for full rank least squares solutions obtained using orthogonalization methods, we substitute the round off error for the perturbations in the condition error equation given above to obtain:

$$\frac{\left\|x-\hat{x}\right\|_2}{\left\|x\right\|_2} \le 2\kappa_2(A)\frac{\left\|b\right\|_2}{\left\|Ax\right\|_2}u + 2\kappa_2(A)^2\frac{\left\|Ax-b\right\|_2}{\left\|Ax\right\|_2}u + 2\kappa_2(A)u .$$

Again, generalization of this procedure to encompass rank deficient problems is discussed in Bjorck.

We do not always care about the agreement between our value for $\hat{x}$ and some theoretical true x. In fact, we often perform these computations to obtain a so called theoretical x in a process called linear regression. In these cases, we are interested in having a low relative residual, and the goodness of fit statistic is most important.

We have all seen two dimensional graphs of data for seemingly stochastic systems where a fit line is drawn through a virtual cloud of scattered data. Fit statistics for these profiles could be in the 500% to 1000% range, yet a clear, predictive function is being revealed and the computational process is a success. Consideration of the specific problem at hand is fundamental in error analysis. Augmenting residual analysis and fit statistics with perturbation experiments is always a good idea.

Furthermore, looking at a problem with a variety of algorithms is frequently quite revealing. For example, although considerably more expensive computationally, orthogonalization methods (including the SVD and column pivoting QR procedures) can be used to solve square systems, resulting in added protection against instability and providing capture for rank deficiency.

Regression Statistics

Linear regression is a most important application of least squares solutions. Here we seek a function that describes the relationships between variables, and enables

prediction of a targeted dependent variable given the values of a set of independent variables. Because of the importance of this application we must discuss certain statistics that relate to a given regression model.

The methods used to obtain the regression equation are exactly those methods we use for the least squares problem. If Ax=b then the formula is $f(x_0, x_1, ...x_n) = y = a_0x_0 + a_1x_1 + ...a_nx_n$. Each column of our A matrix contains values for a specific independent variable. Usually we reserve the first column for evaluating a constant *intercept* value, in which case the column is filled with ones. The b vector is populated with observations of some (presumably) dependent variable associated with each row of independent variables.

The regression function does not necessarily have any theoretical basis in terms of the underlying mechanisms involved in the system being characterized. It simply provides a predictive model for the data. This model has an error associated with it that we will denote ε. The model now has the form $f(x_0, x_1, ...x_n) = y = a_0x_0 + a_1x_1 + ...a_nx_n + \varepsilon$. We assume ε is random and normally distributed with a mean of 0 and a variance of σ^2. The variance can be estimated for a dataset using the equation $\sigma_{est}^2 = s^2 = r^2 / DF$, where r is the 2-norm of the residual and DF is the degrees of freedom. The value for the degrees of freedom is given by $m - n$ ($m - rank$ for solutions obtained using orthogonalizations with column pivoting), where m is the number of data elements in the b vector. The standard deviation of the random error ε is simply $\sigma = \sqrt{\sigma^2}$ and is estimated by $s = \sqrt{s^2}$. This parameter is also referred to as the *standard error of estimate*. The following function returns this value:

```
        public double Standard_error(double[,] A, double[,] b,
double[,] x, double k)
        {
            return Math.Pow(((Math.Pow(Two_norm_residual(A, b, x), 2))
/ ((b.GetLength(0) - 1) - (k + 1))), 0.5);
        }
```

The values we use for x_0, $x_1...x_n$, which we shall call estimators, are themselves estimates. They have a random error associated with them, and that error has a mean of zero. We can gain insight into the contribution to total error from any given estimator by determining the standard deviation of the estimator's sampling distribution. One simple way to accomplish this is to form the diagonal vector of the matrix $(A^TA)^{-1}$. The standard error for the i[th] estimator is given by $s_i = s\sqrt{diag((A^TA)^{-1})_i}$. Here is a function that returns a vector of the standard errors of the estimators:

```
        public double[,] Standard_error_in_estimators(double[,]
A_matrix, double Std_error_estimate)
        {
            double[,] ATAI =
Inverse_by_LU_encoded(Matrix_Multiply(Transpose(A_matrix), A_matrix));
            double[,] result = new double[ATAI.GetLength(0) - 1 + 1,
2];
            int i = 0;
```

```
            for (i = 1; i <= ATAI.GetLength(0) - 1; i++)
            {
                result[i, 1] = (Std_error_estimate) * (Math.Pow(ATAI[i,
i], 0.5));
            }
            return result;
        }
```

Correlation

We need a statistic to evaluate how well the regression model fits the data. The form that is in standard use is the multiple coefficient of correlation. With this statistic a value of 1 is a perfect fit, and a value of zero indicates no relationship between the data and the model. We calculate the statistic as follows:

$R^2 = 1 - \frac{\|residual\|_2^2}{\|b_i - \bar{b}\|_2^2}$ where $\bar{b}$ is the mean of the elements of the b vector. The following function returns this statistic.

```
        public double Correlation_coeff(double[,] A, double[,] b,
double[,] x)
        {
            double numerator = Math.Pow(Two_norm_residual(A, b, x), 2);
            double denominator = 0;
            double[,] Ax = Matrix_Multiply(A, x);
            int m = Ax.GetLength(0) - 1;
            int i = 0;
            double b_Bar = 0;
            for (i = 1; i <= m; i++)
            {
                b_Bar = b[i, 1] + b_Bar;
            }
            b_Bar = b_Bar / System.Convert.ToDouble(m);
            for (i = 1; i <= m; i++)
            {
                denominator = Math.Pow((b[i, 1] - b_Bar), 2) +
denominator;
            }
            return 1 - (numerator / denominator);
        }
```

Confidence and Prediction

Confidence in $f(x_i)$ in the Original Regression Experiment

When we conduct a regression experiment we obtain a function $f(x_0, x_1, ...x_n) = y = a_0x_0 + a_1x_1 + ...a_nx_n$. The function is continuous over the domain in the experimental model. The function also has error and, for any set of a_i values, a calculated value of y has some window of error associated with it. The width of the

window is determined by our confidence level requirements. When we express a value for y we should also include the upper and lower limits for y at a specified confidence level. At a 100% confidence level our results for a given y value will always be $y \pm \infty$, a totally useless result. The tighter our confidence level, the wider the window of error. How do we determine a confidence interval for a given set of values for $a_0, a_1, ...a_n$ at a specified confidence level? We implement the following formula:

$$y \pm t_p s \sqrt{a^T \left(A^T A\right)^{-1} a}$$

where t_p is the t-distribution's positive quantile at a confidence level of $100(1-p)$ and $m-n$ (again, $m-rank$ for rank deficient column pivot orthogonalizations) degrees of freedom (DF); s is the standard error of estimate; and a is the vector of coefficients for which we are evaluating a y result.

This statistic is termed the *confidence interval,* and its use is limited to evaluating calculated values of y for the original dataset used to create the model.

Confidence in a Subsequent Predicted Value for $f(x_i)$ Using the Original Experiment's Regression Model

If we want to use a regression model to predict behavior of a system beyond the original dataset, we need another related, but wider interval — the *prediction interval*. When in doubt, choose this statistic and err on the conservative side. It is determined with the following formula:

$$y \pm t_p s \sqrt{1 + a^T \left(A^T A\right)^{-1} a}$$

First we need a function for obtaining t_p. There are many approaches, including creating a table in code. I like the programmatic approach. Here is a good function attributed to G.W. Hill. It takes as arguments p and DF. We can calculate p with $p = \left(100 - confidence\ level\right)/100$ (e.g., p for a confidence level of 95% is 0.05).

```
        public double Students_tquantile(double P, double n)
        {
            // G. W. Hill Algo 396
            // This algorithm evaluates the positive quantile at the
            // (two-tail) probability level P, for Student's t-
distribution with
            // n degrees of freedom.
            double result = 0;
            if (n == 2)
            {
                result = System.Math.Sqrt(2.0 / (P * (2.0 - P)) - 2.0);
                return result;
            }
            if (n == 1)
            {
                P = P * (System.Math.PI / 2);
```

```
                result = System.Math.Cos(P) / System.Math.Sin(P);
                return result;
            }
            else
            {
                double a = 0, b = 0, c = 0, d = 0, x = 0, y = 0;
                a = 1 / (n - 0.5);
                b = 48 / (Math.Pow(a, 2)); // concern
                c = 20700 * (Math.Pow(a, 3)) / b - 98 * (Math.Pow(a,
2)) - (16 * a) + 96.36;
                d = ((94.5 / (b + c) - 3) / b + 1) * System.Math.Sqrt(a
* System.Math.PI / 2) * n;
                x = d * P;
                y = Math.Pow(x, (2 / n));
                if (y > 0.05 + a)
                {
                    x = SNormInv(P * 0.5);
                    y = Math.Pow(x, 2);
                    if (n < 5)
                    {
                        c = c + 0.3 * (n - 4.5) * (x + 0.6);
                    }
                    c = (((0.05 * d * x - 5.0) * x - 7.0) * x - 2.0) *
x + b + c;
                    y = (((((0.4 * y + 6.3) * y + 36.0) * y + 94.5) / c
- y - 3.0) / b + 1.0) * x;
                    y = a * (Math.Pow(y, 2));
                    if (y > 0.002)
                    {
                        y = System.Math.Exp(y) - 1;
                    }
                    else
                    {
                        y = 0.5 * (Math.Pow(y, 2)) + y;
                    }
                }
                else
                {
                    y = ((1.0 / (((n + 6.0) / (n * y) - 0.089 * d -
0.822) * (n + 2.0) * 3.0) + 0.5 / (n + 4.0)) * y - 1.0) * (n + 1.0) /
(n + 2.0) + 1.0 / y;
                }
                result = System.Math.Sqrt(n * y);
                return result;
            }
        }
```

Students_tquantilecalls the following function (adapted from a VB.NET implementation by Geoffrey C. Barnes, Ph.D., Fels Center of Government and Jerry Lee Center of Criminology, University of Pennsylvania):

```
        public double SNormInv(double p)
        {
            // Geoffrey C. Barnes, Ph.D. Fels Center of Government and
Jerry Lee Center of Criminology University of Pennsylvania, wrote a
VB.NET implementation.
```

```
            // returns a negative normal deviate at the lower tail
probability level p
            double q = 0;
            double r = 0;
            // Coefficients in rational approximations.
            const double A1 = -39.696830286653757;
            const double A2 = 220.9460984245205;
            const double A3 = -275.92851044696869;
            const double A4 = 138.357751867269;
            const double A5 = -30.66479806614716;
            const double A6 = 2.5066282774592392;
            const double B1 = -54.476098798224058;
            const double B2 = 161.58583685804089;
            const double B3 = -155.69897985988661;
            const double B4 = 66.80131188771972;
            const double B5 = -13.280681552885721;
            const double C1 = -0.0077848940024302926;
            const double C2 = -0.32239645804113648;
            const double C3 = -2.4007582771618381;
            const double C4 = -2.5497325393437338;
            const double C5 = 4.3746641414649678;
            const double C6 = 2.9381639826987831;
            const double D1 = 0.0077846957090414622;
            const double D2 = 0.32246712907003983;
            const double D3 = 2.445134137142996;
            const double D4 = 3.7544086619074162;
            // Define break-points.
            const double P_LOW = 0.02425;
            const double P_HIGH = 1 - P_LOW;
            if (p > 0 && p < P_LOW)
            {
                // Rational approximation for lower region.
                q = System.Math.Sqrt(-2 * System.Math.Log(p));
                return (((((C1 * q + C2) * q + C3) * q + C4) * q + C5)
* q + C6) / ((((D1 * q + D2) * q + D3) * q + D4) * q + 1);
            }
            else if (p >= P_LOW && p <= P_HIGH)
            {
                // Rational approximation for central region.
                q = p - 0.5;
                r = q * q;
                return (((((A1 * r + A2) * r + A3) * r + A4) * r + A5)
* r + A6) * q / (((((B1 * r + B2) * r + B3) * r + B4) * r + B5) * r +
1);
            }
            else if (p > P_HIGH && p < 1)
            {
                // Rational approximation for upper region.
                q = System.Math.Sqrt(-2 * System.Math.Log(1 - p));
                return -(((((C1 * q + C2) * q + C3) * q + C4) * q + C5)
* q + C6) / ((((D1 * q + D2) * q + D3) * q + D4) * q + 1);
            }
            else
            {
                throw new ArgumentOutOfRangeException();
            }
        }
```

Next we need a function that takes a, A, s, x and a target confidence level as arguments, and returns y together with the upper and lower limits. Below is a function that performs the task. The predicted y is returned in CIPI(0), the lower and upper confidence limits are returned in CIPI(1) and CIPI(2), and the lower and upper prediction limits are returned in CIPI(3) and CIPI(4).

```
        public double[] Confidence_interval(double[,] a_vector,
double[,] x_vector, double[,] A_matrix, double k, double
standard_error, double Percent_Confinence_level)
        {
            // A 95% confidence interval is the Y-range for a given X
that has a 95% probability
            // for containing the true Y value
            // A 95% prediction interval is the Y range for a given X
where there is a 95% probability
            //  that the next experiment's Y value will occur, based
upon the fit of the present experiment's data
            int i = 0, j = 0;
            int n = A_matrix.GetLength(1) - 1;
            double p = 0;
            p = (100 - Percent_Confinence_level) / 100;
            double t = Students_tquantile(p, (A_matrix.GetLength(0) -
1) - k - 1);
            double y = 0;
            for (i = 1; i <= n; i++)
            {
                y = y + a_vector[1, i] * x_vector[i, 1];
            }
            double[,] ATAI = new double[n + 1, n + 1];
            ATAI =
Inverse_by_LU_encoded(Matrix_Multiply(Transpose(A_matrix), A_matrix));
            double[] aTATAI = new double[n + 1];
            for (i = 1; i <= n; i++)
            {
                for (j = 1; j <= n; j++)
                {
                    aTATAI[i] = a_vector[1, j] * ATAI[j, i] +
aTATAI[i];
                }
            }
            double aTATAIa = 0;
            for (i = 1; i <= n; i++)
            {
                aTATAIa = aTATAIa + aTATAI[i] * a_vector[1, i];
            }
            aTATAIa = Math.Pow(aTATAIa, 0.5);
            double[] CIPI = new double[6]; //  an array to return the
results
            CIPI[0] = (y);
            // Confidence Intervals
            CIPI[1] = (y - (t * standard_error * aTATAIa));
            CIPI[2] = (y + (t * standard_error * aTATAIa));
            // Prediction Intervals
```

```
            double predict = standard_error * (Math.Pow((1 +
(Math.Pow(aTATAIa, 2))), 0.5));
            CIPI[3] = (y - (t * predict));
            CIPI[4] = (y + (t * predict));
            return CIPI;
        }
```

7. Final Organization and Implementation Considerations

Putting the Components Together in a Class Library

Our objective has been to make a class library for matrix computations that we can reference in other applications (as a dll). We have explored all the methods we need for the task, and now we simply need to put them in a class library project. In Visual Studio, create a new C# project, being certain to select the class library option. Name the project *Matrix_Comp_Library_C*. Now insert all of our methods, properties and global dimension statements.

After building the solution, a dll with the name *Matrix_Comp_Library_C .dll* will appear in the bin subdirectory of the class library project. This dll can be copied into the project directory of your application/interface.

Now we need an interface. Create a C# project as usual and include the dll in the project directory. Right click on references, select add a reference, and browse to the dll.

Add `using Matrix_Comp_Library_C;` before the namespace declaration.

Create an instance of the Library class called solver:

```
internal Matrix_Comp_Library_C.Matrix_Comp_Utility solver = new
Matrix_Comp_Library_C.Matrix_Comp_Utility();
```

Now all of the public methods of the class are available to your application project.

Interfacing Considerations

Global Variables

If you need them, consume global variables the moment they are modified, because subsequent calls to other procedures may modify them. For example, the QR_Encoded_P variable is modified by Householder_QR_Pivot, Householder_QR_Pivot_encoded, Givens_QR_Pivot, Givens_QR_Pivot_encoded, HouseholderQR_Pivot_Oneshot, Householder_QR_Pivot_encoded, complete_orthogonal_implicit_Householder, complete_orthogonal_implicit_givens, GivensQR_Pivot_Oneshot, and GivensQR_Pivot_Encoded_Oneshot. We could have written our methods in such a fashion as to force the consumption of these variables, but that is often wasteful.

Entering Matrices

Clearly we want our matrices in two dimensional arrays. It will be most intuitive and compatible with our class library if these arrays are one-based. Getting the data into the arrays is application specific. For example, you might have an analog-to-digital board for transferring measurements to your system. Because most database programs like Excel and many utilities like MatLab™ allow exporting/importing tab and comma delimited files, let us consider an example where we read the data from a tab- or comma-delimited file into an array A by clicking a menu item named OpenA.

At the very top of your form1 code, enter the following using statements:

```
using System.Text.RegularExpressions;
using System.IO;
```

Add an OpenFile Dialog named OpenFile Dialog1 to your form1
Add a SaveFile Dialog named SaveFile Dialog1 to your form1

Add the following global dimensions:

```
public string captionA = "Empty";
public double[,] B;
public string captionB = "Empty";
public double[,] C;
public string captionC = "Empty";
private void OpenA_Click(object sender, EventArgs e)
{
    OpenCsv(ref captionA, ref A);
}

private void OpenCsv(ref string caption, ref double[,] Arr)
{
    try
    {
        int i = 0, j = 0;
        int rows = 0;
        int cols = 0;
        double[,] temparray = new double[2, 2];
        string tempstr1 = "";
        string tempstr2 = "";
        string fileAname = null;
        if (openFileDialog1.ShowDialog() == DialogResult.OK)
        {
            fileAname = openFileDialog1.FileName;
            FileStream fsstream = new FileStream(fileAname,
FileMode.Open, FileAccess.Read);
            StreamReader srreader = new StreamReader(fsstream);
            {
                // determine dimensions of array
                rows = 0;
                cols = 0;
                j = 1;
                i = 1;
                while (srreader.Peek() > -1)
                { // until the end of the file is found
```

```
                            tempstr1 =
System.Convert.ToString(System.Convert.ToChar(srreader.Read())); //
temp storage for readin character
                            if (tempstr1 == "," | char.Parse(tempstr1)
== System.Convert.ToChar(13) | char.Parse(tempstr1) ==
System.Convert.ToChar(9))
                            { // chr(13) is a linefeed
                                if (char.Parse(tempstr1) ==
System.Convert.ToChar(13))// end of row encountered
                                {
                                    cols = j;
                                    j = 0;
                                    i = i + 1;
                                    rows = rows + 1;
                                }
                                j = j + 1;
                                tempstr1 = "";
                                tempstr2 = "";
                            }
                        }
                        srreader.Close();
                        fsstream.Close();
                        srreader = null;
                        fsstream = null;
                        temparray = new double[rows + 1, cols + 1];
                        // data is read one character at a time in row
by row fashion
                        FileStream fsstreamA = new
FileStream(fileAname, FileMode.Open, FileAccess.Read);
                        StreamReader srreaderA = new
StreamReader(fsstreamA);
                        j = 1;
                        i = 1;
                        while (srreaderA.Peek() > -1)
                        { // until the end of the file is found
                            tempstr1 =
System.Convert.ToString(System.Convert.ToChar(srreaderA.Read())); //
temp storage for readin character
                            if (tempstr1 == "," | char.Parse(tempstr1)
== System.Convert.ToChar(13))// chr(13) is a linefeed
                            {
                                temparray[i, j] =
double.Parse(tempstr2);
                                if (char.Parse(tempstr1) ==
System.Convert.ToChar(13))// end of row encountered
                                {
                                    j = 0;
                                    i = i + 1;
                                }
                                j = j + 1;
                                tempstr1 = "";
                                tempstr2 = "";
                            }
                            tempstr2 = tempstr2 + tempstr1; //
concatenation of readin characters to form data elements
                        }
                        srreaderA.Close();
```

```
                        fsstreamA.Close();
                        srreaderA = null;
                        fsstreamA = null;
                        System.GC.Collect();
                        Regex Pathvalue = new Regex(@"(.*)\\");
                        caption =
Pathvalue.Replace(openFileDialog1.FileName, "");
                        Regex extvalue = new Regex(@"\.csv");
                        caption = extvalue.Replace(caption, "");
                        Arr = temparray;
                    }

                }

            }
            catch (Exception ex)
            {
                Interaction.MsgBox(ex.Message,
(Microsoft.VisualBasic.MsgBoxStyle)(0), null);
            }
        }
```

We can save A with a similar logic. The message box implementation requires a reference to Microsoft Visual Basic.

```
        private void SaveA_Click_1(object sender, EventArgs e)
        {
            {
                save_to_csv(A, ref captionA);
            }
        }

        public void save_to_csv(double[,] arr, ref string caption)
        {
            try
            {
                if (saveFileDialog1.ShowDialog() == DialogResult.OK)
                {
                    int R = arr.GetLength(0) - 1;
                    int C = arr.GetLength(1) - 1;
                    int i = 0, j = 0;
                    string temp1 = "";
                    string file_name = saveFileDialog1.FileName;
                    FileStream fsstream = new FileStream(file_name,
FileMode.Create, FileAccess.Write);
                    StreamWriter swwriter = new StreamWriter(fsstream);
                    swwriter.AutoFlush = true;
                    for (i = 1; i <= R; i++)
                    {
                        for (j = 1; j <= C; j++)
                        {
                            temp1 += arr[i, j];
                            if (j < C)
                            {
```

```
                            temp1 += ",";
                        }
                    }
                    swwriter.WriteLine(temp1);
                    temp1 = "";
                }
                swwriter.Close();
                fsstream.Close();
                swwriter = null;
                fsstream = null;
                System.GC.Collect();
                Regex Pathvalue = new Regex(@"(.*)\\");
                file_name = Pathvalue.Replace(file_name, "");
                Regex extvalue = new Regex(@"\.csv");
                file_name = extvalue.Replace(file_name, "");
                caption = file_name;
            }
        }
        catch (Exception ex)
        {
            Interaction.MsgBox(ex.Message,
(Microsoft.VisualBasic.MsgBoxStyle)(0), null);
        }
          }
```

Performing operations on matrices is easy. Here we transpose A into C.

```
        private void transposeAToCToolStripMenuItem_Click(object
sender, EventArgs e)
        {
            try
            {
                if (IsA.Enabled == false)
                {
                    throw new ApplicationException("There is no matrix
in A");
                }
                timedouble = DateTime.Now.ToOADate();
                C = solver.Transpose(A);
                captionC = "(" + captionA + ") Transpose";
                populateC();            }
            catch (Exception ex)
            {
                Interaction.MsgBox(ex.Message,
(Microsoft.VisualBasic.MsgBoxStyle)(0), null);
            }
         }
```

Viewing Matrices

In many applications you may want to view — and perhaps edit — your matrices. The DOT NET DataGridView is just the control we need. I prefer to create a form2 class with a fully docked DataGrid for this purpose. This class should have a global string variable named My_Source_Array to enable us to associate any form2 instance with its corresponding array later. We create an instance of the form2 for each matrix.

```
internal static Form2 Matrix_A;
internal static Form2 Matrix_B;
internal static Form2 Matrix_C;
```

Then we execute a method to populate the grids:

```
private void populateA()
{
    if (!(Matrix_A == null))
    {
        Matrix_A.normalclose = false;
        Matrix_A.Close();
    }
    Matrix_A = new Form2();
    array_to_grid(A, ref Matrix_A.dataGridView1);
    Matrix_A.Text = "Matrix A" + ", name=" + captionA;
    Matrix_A.dataGridView1.Text = captionA;
    Matrix_A.MdiParent = this;
    Matrix_A.Show();
    Matrix_A.Activate();
    Matrix_A.My_Source_Array = "A";
  }
```

that calls the array_to_grid subprocedure.

```
private void array_to_grid(double[,] array, ref DataGridView
dg)
{
    try
    {
        {
            int Rows = array.GetLength(0) - 1;
            int Cols = array.GetLength(1) - 1;
            DataTable dta = new DataTable("matrix");
            DataColumn[] dca = new DataColumn[Cols + 1];
            DataRow[] dra = new DataRow[Rows + 1];
            DataView dv = new DataView(dta);
            dv.AllowNew = false;
            // loop in columns
            int i = 0;
            int j = 0;
            for (j = 0; j <= Cols - 1; j++)
            {
                dca[j] = new DataColumn("");
                dca[j].DataType =
System.Type.GetType("System.Single");
                dta.Columns.Add(dca[j]);
            }
            // create rows
            for (i = 0; i <= Rows - 1; i++)
            {
                dra[i] = dta.NewRow();
                for (j = 0; j <= Cols - 1; j++)
                {
```

```
                            dra[i][j] = array[i + 1, j + 1];
                        }
                    }
                    // add rows to table
                    for (i = 0; i <= Rows - 1; i++)
                    {
                        dta.Rows.Add(dra[i]);
                    }
                    // bind the dataview to the grid
                    dg.DataSource = null;
                    dg.DataSource = dv;
                }
            }
            catch (Exception ex)
            {
                Interaction.MsgBox(ex.Message,
(Microsoft.VisualBasic.MsgBoxStyle)(0), null);
            }
        }
```

Refinements

All of our methods are to be considered starting points in your ongoing process to optimize your applications. The procedures we discussed are meant to be intuitive and easily followed — demonstrating a process and not necessarily establishing best production code practices. There are many areas for improvement, including:

- Implementing methods for iterative refinement in factorization and solution procedures.
- Developing procedures to more effectively handle structured problems like sparse matrices and Vandermonde systems.
- Dealing with matrix storage more efficiently.
- Extending methods to applications involving complex numbers.
- Establishing methods for updating least squares solution through adding or deleting rows in the data.
- Developing methods for solving least squares subject to constraints.
- Develop methods for parallel computation on multi-processor systems.

The list goes on inexhaustibly. A lifetime can be spent exploring (many have) this vast subject area. It is my hope that the reader will use this introduction as a beginning for a more intensive study. I have included a bibliography of works that I found quite useful.

Bibliography

A.A. Anda and H. Park, "Fast Plane Rotations with Dynamic Scaling," *SIAM J. Matrix Anal. Appl. 15*, 1994, pp. 162-174

Bjorck, Ake, *Numerical Methods for Least Squares Problems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1996.

B. D. Bunday, S. M. Husnain Bokhari and K. H. Khan, "A New Algorithm For The Normal Distribution Function," *Test*, v. 6 n. 2, p.369–377, Dec. 1997.

Chan, Tony F., "An Improved Algorithm For Computing The Singular Value Decomposition," *ACM Transactions on Mathematical Software,* Vol. 8, No. 1, March 1982, pp. 72-88.

Golub, Gene H., Charles F. Van Loan, *Matrix Computations 3rd Edition*, The Johns Hopkins University Press, Baltimore, MD, 1996.

Hekstra, Gerben (Philips Research Labs, Eindhoven, The Netherlands), "Evaluation of Fast Rotation Methods," *Journal of VLSI Signal Processing,* Volume 25, 2000, Kluwer Academic Publishers, The Netherlands, pp. 113–124.

Hill, G. W. (C.S.I.R.O., Division of Mathematical Statistics, Glen Osmond, South Australia), "Algorithm 395 Student's T-Distribution," *Communications of the ACM,* Volume 13, Number 10, October, 1970, L. D. Fosdick, Editor, p. 617.

Pollock, D.S.G., *A Handbook of Time-Series Analysis, Signal Processing and Dynamics,* Queen Mary and Westfield College, The University of London, Academic Press, 1999.

Serber, George A. F., Alan J. Lee, *Linear Regression Analysis, 2nd Edition,* Wiley and Sons, Inc., Hoboken, NJ, 2003.

Stewart, G.W., "The Economical Storage of Plane Rotations," *Numer. Math. 2,* 1976, Springer-Verlag, pp. 37–38.

Watkins, David S., *Fundamentals of Matrix Computations, 2nd Edition*, John Wiley and Sons, Inc., New York, NY, 2002.

Wrede, Robert C., *Introduction to Vector and Tensor Analysis,* Dover Publications, NY, 1972.

http://mathworld.wolfram.com/ — A Wolfram Web Resource, Wolfram Research, Inc., 100 Trade Center Drive Champaign, IL 61820-7237, USA

Statistical data analysis by Dr. Dang Quang A and Dr. Bui, The Hong Institute of Information Technology Hoang Quoc Viet road, Cau Giay, Hanoi