California State University Northridge

College of Engineering and Computer Science
Mechanical Engineering Department
Engineering Analysis Notes

Last updated: September 14, 2017 Larry Caretto

Numerical Solutions of Simultaneous Linear Equations

Introduction

The general approach to solving simultaneous linear equations is known as Gauss elimination. There are several variants of the basic technique; however, all start with the basic idea of reducing a set of equations to an upper triangular form that can be easily solved. Two important parts of this process are (1) the detection of a system of equations that does not have a unique solution and (2) reducing round-off error in the calculations. These notes do not cover iterative methods for the solution of equations. These methods will be discussed as part of the consideration of sparse matrices that arise in the numerical solution of partial differential equations.

Although we are used to writing equations as the form $3x + 2y + 6z = 3$, etc., the computer solutions are based on a more general notation. Indeed, we will drop all reference to writing the equations as such and simply use the data in the equations. In particular, we can write the system of equations in matrix notation as $\mathbf{Ax} = \mathbf{b}$. We will develop a set of algorithms that uses the data in the $\mathbf{A}$ matrix and the $\mathbf{b}$ vector to obtain the solutions in the $\mathbf{x}$ vector, without reference to actual algebraic equations.

A key feature of algorithms for solving simultaneous equations is called a “pivoting strategy” that is used to reduce round-off error that occurs in the solutions of these equations. These notes begin with background material on computer data representation to give background on the topic of round-off error.

Computer representation of data

Because of the binary representation of data in computers, numbers are not represented exactly. This is no different than the custom that we practice when we represent π as 3.14, 3.14159, 3.14159265358979, or some other number depending on the accuracy we require in a given calculation. Although the binary representation of numbers in a computer uses base two instead of our base ten, the computer designer’s decision on the number of significant figures used to represent an inexact number limits the accuracy in our calculations.

Numbers in computers are typically represented as integer data types, used for counting, and floating point data types used for computations. Although the names for the data types differ in different languages, the floating-point data types can be thought of as single precision, double precision, and extended precision. Each of these types uses more bits to represent the number. In a typical computer 32 bits (4 bytes) are used for single precision, 64 bits (8 bytes) for double precision, and 80 bits (10 bytes) for extended precision. Standards for computing languages give wide options to compiler vendors and the word size for the different types may be different for different machines and even for different compilers on the same machine.

The structure of a floating-point data type represents the number in a binary exponential form, $\pm m \bullet 2^c$, where c is the characteristic (or exponent) and m is the mantissa. This is like our usual representation of numbers as 6.023×10^{23}, but it uses 2 instead of 10 for the exponent base. As more bits are used for the word, both the range of the powers available in the characteristic and the number of significant digits in the mantissa increase. For the typical single-precision floating-point data type, the magnitude of the numbers can range between 10^{-38} and 10^{38}, and the mantissa can represent about 7 significant figures. A typical double-precision type can range between 10^{-308} and 10^{308}, and the mantissa can represent more than 15 significant figures.

The “machine epsilon” is the name given to the largest number such that 1 + (machine epsilon) = 1 in the computer. For a 4-byte single precision this value is 1.19×10^{-7}, and for an 8-byte double precision, the machine epsilon is 2.22×10^{-16}. This shows us that the single-precision cannot quite represent seven significant figures (otherwise the machine epsilon would be less than 10^{-7}) while the double precision can easily represent 15 significant figures. Although single precision may seem to have significant accuracy for most engineering calculations, it is possible that rounding that occurs in a series of calculations can increase the difference between the true solution and the one obtained in a computer. This is particularly true in the solution of simultaneous algebraic equations, and algorithms have been devised to reduce this round-off error. See the additional information on round-off error in Appendix A and binary numbers in Appendix C.

Simultaneous Linear Algebraic Equations

In order to solve a set of simultaneous linear equations we use a process that replaces equations in the set by equivalent equations. We can replace any equation in the set by a linear combination of other equations without changing the solution of the system of equations. For example, consider the simple set of two equations with two unknowns. Here we use the notation x_1 and x_2 in place of the more conventional x and y for our two unknowns. This allows us to generalize our processes to equation sets of any size and to introduce matrix notation.

$$\begin{aligned} 3x_1 + 5x_2 &= 13 \\ 7x_1 - 4x_2 &= -1 \end{aligned} \qquad [1]$$

You can confirm that $x_1 = 1$ and $x_2 = 2$ is a solution to this set of equations. To solve this set of equations we can replace the second equation by a new equation, which is a linear combination of the two equations without changing the solution. The particular combination we seek is one that will eliminate x_1. We can do this by subtracting the first equation, multiplied by 7/3, from the second equation to obtain the following pair of equations, which is equivalent to the original set in equation [1].

$$\begin{aligned} 3x_1 + 5x_2 &= 13 \\ -\frac{47}{3}x_2 &= -\frac{94}{3} \end{aligned} \qquad [2]$$

We can readily solve the second equation to find $x_2 = 2$, and substitute this value of x_2 into the first equation to find that $x_1 = [13 - 5(2)]/3 = 1$. The same process can be applied to the solution of three equations in three unknowns. Consider the following set of equations.

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \qquad (A) \\ -3x_1 + 2x_2 + \quad 9x_3 &= \quad 13 \qquad (B) \\ 7x_1 + 3x_2 + \quad 8x_3 &= \quad 14 \qquad (C) \end{aligned} \qquad [3]$$

The general procedure for combining two equations, A and B, to eliminate the x_k term from equation B is described below. The description uses equation set [3] above as an example showing how to combine the first equation (A) and the second equation (B) to eliminate the x_1 coefficient from the second equation.

- **Multiply equation A by the ratio of the x_k coefficient in equation B to the x_k coefficient in equation A.** For example, multiply the first equation in [3] by –3/2 to obtain the intermediate result $-3x_1 + 6x_2 + 39x_3 = 51$, as the first step in eliminating the x_1 coefficient in the second equation.
- **Subtract the equation found in the previous step from equation B. The result will be a new equation in which the x_k coefficient is zero.** In the example we subtract $-3x_1 + 6x_2 + 39x_3 = 51$ from $-3x_1 + 2x_2 + 9x_3 = 13$ to give the result $-4x_2 - 30x_3 = -38$.
- **Replace equation B by the equation found in the previous step.** In the example, the new second equation, $-4x_2 - 30x_3 = -38$, has a zero coefficient for x_1.

In practice, all three steps outlined above are combined into a single set of operations. We can summarize all the work in the example above (multiplying the first (A) equation by –3/2 and subtracting the result from the second (B) equation) by the following set of operations. This gives us our new second equation whose x_1 coefficient is zero.

$$\left[-3-\left(\frac{-3}{2}\right)2\right]x_1+\left[2-\left(\frac{-3}{2}\right)(-4)\right]x_2+\left[9-\left(\frac{-3}{2}\right)(-26)\right]x_3=\left[13-\left(\frac{-3}{2}\right)(-34)\right] \quad [4]$$

In a similar way, we can set the coefficient of x_3 in the third equation equal to zero by subtracting 7/2 times the first equation from the third equation as follows.

$$\left[7-\left(\frac{7}{2}\right)2\right]x_1+\left[3-\left(\frac{7}{2}\right)(-4)\right]x_2+\left[8-\left(\frac{7}{2}\right)(-26)\right]x_3=\left[14-\left(\frac{7}{2}\right)(-34)\right] \quad [5]$$

Doing the indicated arithmetic in equations [4] and [5] and using these results to replace the second and third equations in [3] gives

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \\ 0x_1 - 4x_2 - 30x_3 &= -38 \\ 0x_1 + 17x_2 + 99x_3 &= 133 \end{aligned} \quad [6]$$

We see that we have reduced the last two equations to two equations in two unknowns and we can make the x_2 coefficient in the third equation zero by multiplying the second equation by 17/(-4) and subtracting the result from the third equation. (Since the x_1 coefficients in both equations are zero, the result of the subtraction will have a zero for both the x_1 and x_2 coefficient.)

$$\left[0-\left(\frac{17}{-4}\right)0\right]x_1+\left[17-\left(\frac{17}{-4}\right)(-4)\right]x_2+\left[99-\left(\frac{17}{-4}\right)(-30)\right]x_3=\left[133-\left(\frac{17}{-4}\right)(-38)\right] \quad [7]$$

We can use equation [7] to replace the third equation in [6], giving the following set of equations.

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \\ 0x_1 - 4x_2 - 30x_3 &= -38 \\ 0x_1 + 0x_2 - \frac{57}{2}x_3 &= -\frac{57}{2} \end{aligned} \qquad [8]$$

This set of equations should have the same solution as the original set of equations in [3], since none of our operations have changed the values of the unknowns. We say that this set of equations is equivalent to the original set of equations in that it has the same set of solutions.

We see that we can easily solve the third equation for $x_3 = (-57/2) / (-57/2) = 1$. Once we know that $x_3 = 1$, we can substitute this result into the second equation and solve for x_2.

$$x_2 = \frac{-38 - (-30)x_3}{(-4)} = \frac{-38 - (-30)(1)}{(-4)} = \frac{-8}{(-4)} = 2 \qquad [9]$$

We now know both x_2 and x_3 and we can substitute these values into the first equation to obtain x_1.

$$x_1 = \frac{-34 - (-26)x_3 - (-4)x_2}{2} = \frac{-34 - (-26)(1) - (-4)(2)}{2} = \frac{0}{2} = 0 \qquad [10]$$

You can verify that the solution $x_1 = 0$, $x_2 = 2$, and $x_3 = 1$ satisfies the original set of equations in [3].

We have seen how to solve two equations in two unknowns and three equations in three unknowns. We want to generalize the process to solve n equations in n unknowns. To do this, we introduce the following notation for this general case: the coefficient of x_j in equation i is written as a_{ij}. The right-hand side of equation i is written as b_i. With this notation we write our general equation as follows

$$\sum_{j=1}^{m} a_{ij} x_j = b_i \qquad i = 1, \ldots, n \qquad [11]$$

In matrix notation, we would define an **A** matrix whose coefficients are $[a_{ij}]$, a right-hand side vector, **b**, and a solution vector, **x**. These are written out in full below.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & \cdots & a_{nn} \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} \qquad [12]$$

With these matrix definitions, we write our system of equations as **Ax** = **b**.

The basic process to solve this system of equations is known as Gauss elimination. The goal of this process is to reduce the original set of equations into an equivalent set in which there is only one unknown in the last equations, two unknowns in the next to last equation, three unknowns in the last equation but two, and so forth, until only the first equation contains all the unknowns. The operations that we use to do this do not change the solutions. However the final result, which is called an upper-triangular form, can be readily solved for all the unknowns. This final, upper-triangular form, in matrix notation, is shown below.

$$\begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \cdots & \cdots & \alpha_{1n-1} & \alpha_{1n} \\ 0 & \alpha_{22} & \alpha_{23} & \cdots & \cdots & \alpha_{2n-1} & \alpha_{2n} \\ 0 & 0 & \alpha_{33} & \cdots & \cdots & \alpha_{3n-1} & \alpha_{3n} \\ \vdots & \vdots & \vdots & \ddots & & & \\ \vdots & \vdots & \vdots & & \ddots & & \\ 0 & 0 & 0 & \cdots & \cdots & \alpha_{n-1n-1} & \alpha_{n-1n} \\ 0 & 0 & 0 & \cdots & \cdots & 0 & \alpha_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \vdots \\ \beta_{n-1} \\ \beta_n \end{bmatrix} \qquad [13]$$

In order to have the same solutions, all the operations are done on the complete equation; this means that we change not only the coefficients in the **A** matrix, but also the coefficients in the right-hand-side **b** vector. (The same operations that are used to obtain the revised coefficient matrix are used to obtain the revised right-hand-side vector.)

The revised **A** and **b** matrices, shown in equation [13] are obtained in a series of steps. This is a generalization of the process used above for two and three equations. (In the following discussion, we talk about operations on rows and operations on equations. These mean the same thing. Operations on rows of the **A** matrix and similar operations on rows of the **b** vector are equivalent to operations on the equations represented by the matrix coefficients.) In the first step, the x_1 coefficients are eliminated from all equations except the first one. Equations [4] and [5] show numerical examples of this process. To generalize the process, we say that the first row is multiplied by a factor and subtracted from row i in such a way that a_{i1}, the coefficient of x_1 in row i is set to zero. The resulting equation is then used in place of equation i. This is repeated for each row below the first row. In order for this subtraction to set a_{i1} to zero, we have to multiply the first equation by (a_{i1}/a_{11}). We must apply this same subtraction process to all terms in row i, including the b_i term. This operation, which can be represented by the following replacement operations,[1] is done on the coefficients in rows (equations) 2 to n.

$$\left(a_{ij} \leftarrow a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j} \quad j = 1, \ldots n \quad and \quad b_i \leftarrow b_i - \frac{a_{i1}}{a_{11}} b_1 \right) \quad i = 2, \ldots n \qquad [14]$$

After equation [14] is applied, the only nonzero x_1 coefficient is in the first equation (represented by the first row of the **A** matrix and the **b** vector.) You can confirm that this will set $a_{i1} = 0$ for $i > 1$. You can also apply the formulae in [14] to the system of equations in [3] to obtain equations [4] and [5]. The elimination process is next applied to make the x_2 coefficients on all equations below the second equation zero. Here we use the same process. We multiply the second row

[1] Here we use the computer pseudocode notation $a_{1j} \leftarrow$ <*expression*> to show that the old value of a_{1j} is replaced by the value computed in the <*expression*>, where the <*expression*> may contain the old value of a_{1j}. This avoids the requirement in mathematical notation to distinguish between the old and new values of a_{ij} in an equation.

(equation) by the factor a_{i2}/a_{22} and subtract it from row i, where i ranges from 3 to n, and use the result to replace row i. The replacement operations to accomplish this are shown below.

$$\left(a_{ij} \leftarrow a_{ij} - \frac{a_{i2}}{a_{22}} a_{2j} \quad j = 2, \ldots n \quad and \quad b_i \leftarrow b_i - \frac{a_{i2}}{a_{22}} b_2 \right) \quad i = 3, \ldots n \qquad [15]$$

Equation [15] has the same form as equation [14]; only the fixed subscripts use row 2 instead of row 1 and the starting point for the row and column operations are different. Again, you can apply equation [15] to the system of equations in [6] to obtain equation [7].

The process described by equations [14] and [15] can be continued until the upper-triangular form shown in equation [13] is obtained. We can develop a general form for this operation that can be used in a computer code. The equation that is subtracted from all equations below it, at any step in the process, is called the pivot equation (or the pivot row in the matrix). The goal of the subtraction process is to set $a_{row,pivot}$, the coefficient of x_{pivot}, in each row below the pivot row to zero. To do this we (1) multiply the pivot row by $a_{row,pivot}/a_{pivot,pivot}$, (2) subtract it from the given row, and (3) use the result to replace the given row. This gives the following generalization of equations [14] and [15], where we use the index k to denote the pivot row.

$$\left(a_{ij} \leftarrow a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj} \quad j = k, \ldots n \quad and \quad b_i \leftarrow b_i - \frac{a_{ik}}{a_{kk}} b_k \right) \quad i = k+1, \ldots n \qquad [16]$$

The replacement operations in [16] are applied using all rows except the last row as pivot rows. This process will produce the upper-triangular form shown in equation [13]. From this form, we can easily find all the values of x_i. The process used to find these unknowns from the upper triangular matrix is called back substitution, because the unknowns are found in reverse order. From equation [13] we see that we can simply find x_n as β_n/α_{nn}. Once x_n is known, we find $x_{n-1} = [\beta_{n-1} - \alpha_{n-1,n}x_n]/\alpha_{n-1,n-1}$. The general equation for this back-substitution process, inferred from equation [13]. is shown below.

$$x_i = \frac{\beta_i - \sum_{j=i+1}^{n} \alpha_{ij} x_j}{\alpha_{ii}} \qquad i = n-1, n-2, \ldots, 1 \qquad [17]$$

When we are solving for x_i, all previous values of x_j required in the summation are known.

The C++ code below shows a simplified version[2] of how this method is applied to the solution of equations. (We assume that all data values are declared and initialized.) The number of equations is equal to the number of unknowns, n. The first of three loops used to get the upper triangular array, is the loop that uses all rows (except the last row) as the pivot row. Here the program variable, pivot, is used as the subscript for this row. The code execution is simplified by augmenting the a matrix so that $a_{i,n+1} = b_i$. This allows the code to proceed without separate consideration of similar operations on the **A** and **b** matrix components.

```
          // augment column n+1 of a matrix with b values
     for ( row = 1; row <=n; row++) a[row][n+1] = b[row];
          // get upper triangular array
```

[2] As discussed below, actual code would have to account for the possibility that the system of equations might not have a solution. It would also use different operations to reduce roundoff error.

```
for (pivot = 1; pivot < n; pivot++ )
   for ( row = pivot+1; row <= n; row++ )
      for ( column = row+1; column <= n+1; column++)
         a[row][column] -= a[row][pivot] * a[pivot] [column]
                         / a[pivot][pivot];

     // Upper triangular matrix complete; get x values

for (row = n; row >= 1; row--)
{
   x[row] = a[row][n+1];
   for ( column = n; column < row; column-- )
         x[row] -= a[row][column] * x[column];
   x[row] /= a[row][row];
}
```

The process outlined above for the solution of a set of simultaneous equations is known as the Gaussian elimination procedure. Alternative procedures such as the Gauss-Jordan method and LU decomposition, work in a similar manner. They produce an upper triangular matrix or diagonal matrix that is then used to solve for the values of x_i in reverse order.

Detecting equations with no solution

If the solution process outlined above is used on certain matrices, it may not be possible to obtain a unique solution. Consider the two sets of equations shown below.

$$\begin{matrix} 3x_1 + 5x_2 = 13 \\ 6x_1 + 10x_2 = 26 \end{matrix} \qquad \textit{and} \qquad \begin{matrix} 3x_1 + 5x_2 = 13 \\ 6x_1 + 10x_2 = 27 \end{matrix} \qquad [18]$$

In the set of equations on the left, the second equation is simply twice the first equation. If we multiply the first equation by two and subtract it from the second equation, we get the result that 0 = 0. Thus, the second equation gives us no new information on the relationship between x_1 and x_2. We say that this system of equations has an infinite number of solutions. Any value of $x_2 = 2.6 – 0.6x_1$ will satisfy both equations. The second set of equations has no solutions. If we multiply the first equation by two and subtract it from the second equation, we have the result that 0 = 1! Thus, this second set of equations is incompatible and does not have a solution.[3]

This simple example shows a general result. When the Gaussian elimination procedure produces a row of the **A** matrix that is all zeros, we will not have a unique solution to our problem. If the corresponding row of the **b** vector is zero, we will have an infinite number of solutions; if the **b** vector row is not zero, the system of equation has no solution. In each case, the **A** matrix has no inverse and is called a singular matrix. We cannot find a unique solution to **Ax** = **b** when we have a singular matrix. In a singular matrix, there is a linear dependence among the rows. Since the Gauss elimination process is a set of linear operations on the rows of the matrix, it can be used to detect linear dependence in the rows of a matrix.

We detect a singular matrix by looking for zeros in the pivot column. This is the column which contains the diagonal term, $a_{pivot,pivot}$ on the pivot row. Just finding one zero on the pivot column

[3] This result has a geometric interpretation. When we have two simultaneous linear algebraic equations, we can plot each equation in x_1–x_2 space. The solution to the pair of equations is located at the point where both equations intersect. If we did this for the left set of equations in [18], we would only have a single line. There is no second equation to give us an intersection point. Any point along the single line satisfies the pair of equations. If we plot the pair of equations on the right, we would get parallel lines that never intersect. Thus, there is no intersection to provide a solution.

does not mean the matrix is singular. Consider the following set of equations, which is similar to equation [3] except that a_{22} and b_2 have been changed.

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \\ -3x_1 + 6x_2 + 9x_3 &= 21 \\ 7x_1 + 3x_2 + 8x_3 &= 14 \end{aligned} \qquad [19]$$

Applying the same steps we used in equations [4] and [5] gives the follow result in place of equation [6].

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \\ 0x_1 + 0x_2 - 30x_3 &= -30 \\ 0x_1 + 17x_2 + 99x_3 &= 133 \end{aligned} \qquad [20]$$

Here we see that that $a_{22} = 0$, but Gauss elimination wants to divide by this element. Although we cannot divide by zero, we can solve the problem by swapping the second and third equations. This process will have not affect the results since we can place our equations in any order. Once we swap the two equations in this simple example, we have our upper triangular form that we can solve for the solutions.

Consider the following set of equations. This is a further modification of the set of equations in [18] in which the third equation has been changed.

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \\ -3x_1 + 6x_2 + 9x_3 &= 21 \\ -1x_1 + 2x_2 - 17x_3 &= -13 \end{aligned} \qquad [21]$$

When we apply Gauss elimination, using the first row as the pivot row, we get the following result.

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \\ 0x_1 + 0x_2 - 30x_3 &= -30 \\ 0x_1 + 0x_2 - 30x_3 &= -30 \end{aligned} \qquad [22]$$

In this case, we cannot swap rows two and the three to get a nonzero value of a_{22}. We see that both of these equations give us $x_3 = 1$ and we are left with one equation (the first equation) in two unknowns. Thus, the set of equations in [21] has an infinite number of solutions. For any arbitrary constant, α, $x_3 = 1$, $x_2 = \alpha$, and , $x_1 = 2\alpha - 4$, will provide a solution to the set of equations in [21]. We see that this occurs because the third equation is a linear combination of the first two. (In this example, equation three simply the sum of the first two equations.) If the right-hand side of the third equation in [21] had been an arbitrary number K, in place of –13, we would have the following result in place of [22].

$$\begin{aligned} 2x_1 - 4x_2 - 26x_3 &= -34 \\ 0x_1 + 0x_2 - 30x_3 &= -30 \\ 0x_1 + 0x_2 - 30x_3 &= \left[K - \left(\frac{-1}{2}\right)(-34)\right] \end{aligned} \qquad [23]$$

We can subtract the second equation from the third equation to the result that 0 = (K + 13)/(-30). This can only be true if K = -13. A set of equations with K having any other value cannot have a solution. The two cases shown in equations [19] and [21] are ones in which the **A** matrix is singular. It does not possess an inverse because the rows of the matrix are linearly dependent. In such cases, we cannot find a *unique* solution to the equations.

The detection of a singular matrix arises naturally in the Gauss elimination process. We have to swap equations when we find a zero as the diagonal element in the pivot row. If we cannot find a nonzero element in the column that contains this element, called the pivot column, in equations below the pivot row, we know that we have a singular matrix. In practical applications, we test for a number that is small compared to round-off error rather than testing for zero. Having all zeros in the pivot column means that we can use the other columns to obtain zeros in all columns in the last row of the matrix. Thus, we produce a row of all zeros, which is an indication of a singular matrix.

Reducing round-off error

Gauss elimination involves many calculations, particularly for large system of equations. In additions, some sets of equations may be ill conditioned. "Ill-conditioned" is the name given to systems in which the matrix is nearly linearly independent, but not exactly so. Conventional computer codes for Gauss elimination and its variants use some form of row or column exchange to minimize round-off error. Exchange of rows is straightforward, but exchange of columns changes the identity of the variables and is more difficult to implement. The extra effort for computations with column pivoting is usually not worth the effort and the general approach used, called scaled partial pivoting, is described below.

Initially a scale factor is found for each equation. This factor is the maximum element in absolute value on the left-hand side of the matrix. If each equation is divided by its scale factor, then the maximum coefficient in each equation is 1. Before using any row of the matrix as the pivot row, all rows from the pivot row to the last row in the matrix are examined to find the row with the maximum (scaled absolute value) element in the pivot column. This row is then swapped with the pivot row. The process of using one row as the pivot row then proceeds in the usual fashion.

Condition of a matrix

In numerical analysis, a problem is called ill conditioned if a small relative change in input produces a large relative change in the results. In general, then, the condition number for any problem can be defined as follows.

$$Condition = \left| \frac{\dfrac{Change\ in\ result}{result}}{\dfrac{Change\ in\ input}{input}} \right| \qquad [24]$$

For calculations with vectors and matrices, we use the norms to obtain relationships about the condition of a problem. We have to first discuss how we compute the norm of a matrix in such a way that it is compatible with the norm of a vector.

Computing matrix norms – The norm of a matrix may be computed in various ways. The simplest way, which is an extension of the concept of the length of a vector, is called the Frobenius norm. This is defined as follows.

$$\|\mathbf{A}\|_{Frobenius} = \sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}^2 \qquad [25]$$

The norm of a square matrix, **A**, with n rows and n columns, can also be defined in terms of the norm of a vector. This is based on the idea that if **x** is a column vector with n rows then **Ax** is also a vector with n rows. Thus, we can define a vector norm for both ||**Ax**|| and ||**x**||. For any **x** which is not **0**, so that ||**x**|| ≠ 0, the ratio of these can be interpreted as a vector norm for the matrix, **A**. In particular, we define this norm as follows:

$$\|\mathbf{A}\| = \max_{\|\mathbf{x}\|} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \qquad [26]$$

Equation [26] tells us that we seek the vector, **x**, which produces the maximum value of the ratio in equation [26], provided that **x** is not the null vector, **0**. The actual value of the matrix norm defined according to equation [26] depends on the definition that we choose for the vector norm. As noted in Kreyszig, it is possible to show that equation [26] leads to the following definition of a matrix norm if we choose the "one" norm (sum of absolute values of the vector components).

$$\|\mathbf{A}\| = \max_{j} \sum_{i=1}^{n} |a_{ij}| \qquad [27]$$

This is called the column sum norm because we take the sum of the absolute values of the elements in each column. We then pick the maximum of these column sums as the norm. We can also show that choosing the "infinity" norm as our definition of a norm gives the following, row sum, definition of the matrix norm.

$$\|\mathbf{A}\| = \max_{i} \sum_{j=1}^{n} |a_{ij}| \qquad [28]$$

Regardless of the definition of the matrix norm, the basic form of the definition in equation [26] tells us that ||**A**|| ||**x**|| = max$_{||x||}${ ||**Ax**|| }. Since any value of ||**Ax**|| must be less than the maximum possible value of ||**Ax**||, we can say that ||**Ax**|| ≤ max$_{||x||}${ ||**Ax**|| } = ||**A**|| ||**x**||. This means that the following inequality must always hold.

$$\|\mathbf{Ax}\| \le \|\mathbf{A}\|\,\|\mathbf{x}\| \qquad [29]$$

Condition number definition – We denote a numerical solution to **Ax** = **b** that has errors as $\tilde{\mathbf{x}}$. This incorrect solution leads to a residual difference, **r**, which is defined as follows.

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\tilde{\mathbf{x}} = \mathbf{Ax} - \mathbf{A}\tilde{\mathbf{x}} = \mathbf{A}(\mathbf{x} - \tilde{\mathbf{x}}) \qquad [30]$$

We want to see the effect that this residual has on the relative error in the solution.

If we premultiply this equation by $\mathbf{A}^{-1}$, we obtain an equation for the difference between the correct and incorrect solution vectors. We then take the norm of this difference and use the inequality in [29] ($||\mathbf{Ab}|| \le ||\mathbf{a}||\,||\mathbf{b}||$) to obtain the following result.

$$(\mathbf{x}-\tilde{\mathbf{x}}) = \mathbf{A}^{-1}\mathbf{r} \qquad \Rightarrow \qquad \|\mathbf{x}-\tilde{\mathbf{x}}\| = \|\mathbf{A}^{-1}\mathbf{r}\| \le \|\mathbf{A}^{-1}\|\|\mathbf{r}\| \tag{31}$$

We can also apply the result that $||\mathbf{Ab}|| \le ||\mathbf{a}||\,||\mathbf{b}||$ to $\mathbf{Ax} = \mathbf{b}$ as follows.

$$\|\mathbf{b}\| = \|\mathbf{Ax}\| \le \|\mathbf{A}\|\|\mathbf{x}\| \qquad \Rightarrow \qquad \frac{1}{\|\mathbf{x}\|} \le \frac{\|\mathbf{A}\|}{\|\mathbf{b}\|} \tag{32}$$

If we divide the inequality in [31] by $||\mathbf{x}||$ we obtain

$$\frac{\|\mathbf{x}-\tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \|\mathbf{A}^{-1}\|\|\mathbf{r}\|\frac{1}{\|\mathbf{x}\|} \tag{33}$$

We now use a basic rule for inequalities: if u < cv, where c is positive, and v < w, then u < cw. We use this rule to combine the inequalities in [32] and [33] so as to eliminate the $1/||\mathbf{x}||$ term on the right-hand side of [33].

$$\frac{\|\mathbf{x}-\tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \|\mathbf{A}^{-1}\|\|\mathbf{r}\|\frac{1}{\|\mathbf{x}\|} \le \|\mathbf{A}^{-1}\|\|\mathbf{r}\|\frac{\|\mathbf{A}\|}{\|\mathbf{b}\|} \tag{34}$$

We solve for the relative error in $\mathbf{x}$ in terms of the ratio the norm of the residual to the norm of the right-hand-side vector, $\mathbf{b}$.

$$\frac{\dfrac{\|\mathbf{x}-\tilde{\mathbf{x}}\|}{\|\mathbf{x}\|}}{\dfrac{\|\mathbf{r}\|}{\|\mathbf{b}\|}} \le \|\mathbf{A}^{-1}\|\|\mathbf{A}\| \equiv \kappa(\mathbf{A}) \tag{35}$$

In this equation we have defined the ***condition number***, $\kappa(\mathbf{A})$, of the matrix, $\mathbf{A}$, as the product of the norms, $||\mathbf{A}||\,||\mathbf{A}^{-1}||$. Equation [35] tells us that the relative error in the solution vector, $\mathbf{x}$, is proportional to the condition number times the relative residual. A large condition number indicates that there may be difficulties in obtaining an accurate solution. There is no magic value for a condition number that indicates a potential problem with solving linear equations. Generally a condition number greater than 100 indicates a possible problem.

Sample computation of condition number – Consider the following matrix.

$$\mathbf{A} = \begin{bmatrix} 1.00001 & 0.99999 \\ 1 & 1 \end{bmatrix} \tag{36}$$

We see that the two rows of this matrix are almost, but not quite, the same. When we are close to having a linear dependence in the rows of a matrix, we have an ill conditioned matrix. To determine the condition number, we first find the inverse using the general formula for the inverse of a (2 x 2) matrix shown below.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^{-1} = \frac{1}{a_{11}a_{22} - a_{21}a_{12}} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} \quad [37]$$

Applying this formula to the matrix of equation [36] gives the inverse of that matrix.

$$\mathbf{A}^{-1} = \begin{bmatrix} 50000 & -49999.5 \\ -50000 & 50000.5 \end{bmatrix} \quad [38]$$

The norms for the **A** and **A**$^{-1}$ matrices are computed in the table below. Both the row-sum (infinity) norm and the column sum (one) norms are computed.

A calculations				**A**$^{-1}$ calculations			
Row sums		Column sums		Row sums		Column sums	
Row 1	Row 2	Column 1	Column 2	Row 1	Row 2	Column 1	Column 2
2	2	2.00001	1.99999	99999.5	100000.5	100000	100000
$\lVert\mathbf{A}\rVert_\infty = 2$		$\lVert\mathbf{A}\rVert_1 = 2.00001$		$\lVert\mathbf{A}^{-1}\rVert_\infty = 100000.5$		$\lVert\mathbf{A}^{-1}\rVert_1 = 100000$	

In this example both choices of a norm give a condition number of 200,001, indicating that the matrix is very ill conditioned. We expected this result since we could see that the two equations implied by the original **A** matrix were very nearly identical. In a larger matrix, we can have linear dependencies that are not so obvious. In such cases, the condition number will indicate possible problems with the solution of equations based on those matrices.

The condition number also enters into the measure of the effects of changes in the input data in **A** or **b** on changes in the solution. Appendix B shows how the condition number enters into these results.

The Gauss-Jordan method

This is a variant of Gauss elimination in which the pivot row is subtracted from not only the rows below the pivot row, but also from the rows above it. (Of course, the pivot row is multiplied by a suitable factor to obtain zeros in the pivot column in all those rows.) With this change, the **A** matrix is converted to a diagonal matrix, so the solutions for **x** are simply the modified values in the **b** vector, divided by the diagonal elements on the row. The Gauss-Jordan method commonly divides each pivot row by its diagonal element so that the solutions are simply the elements of the modified **b** vector. This method is not commonly used because it involves more computational work than Gauss elimination. It is sometimes used in the computation of matrix inverses.

If an inverse matrix is required, which is an unusual case, it can be found by the Gauss elimination approach or one of the variants of that approach. If we define **B** = **A**$^{-1}$, we have **AB** = **AA**$^{-1}$ = **I**. The matrices in the formula for **AB** = **I** are shown below.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots & \cdots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \cdots & \cdots & b_{2n} \\ b_{31} & b_{32} & b_{33} & \cdots & \cdots & b_{3n} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ b_{n1} & b_{n2} & b_{n3} & \cdots & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & 1 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \cdots & 1 \end{bmatrix}$$
[39]

This form as well as the usual equation for matrix multiplication, written for equation [39] below, show that an individual column of the matrix **B** = **A**$^{-1}$ is found from the same process used to solve **Ax** = **b**. To find column j of the inverse, the right-hand-side vector, **b** is all zeros except for row j, which is a one. The column vector with the solution, **x**, is the jth column of the inverse.

$$\mathbf{AB} = \mathbf{I} \quad \Rightarrow \quad \sum_{k=1}^{n} a_{ik} b_{kj} = \delta_{ij} \qquad [40]$$

Because the Gauss-Jordan method is often applied to the computation of matrix inverses, the code for this method is usually written to solve **AX** = **B**. In this notation, **B** is a matrix with one or more columns corresponding to one or more right hand side (**b**) vectors and **X** is a matrix that contains the solutions for each of those **b** vectors.

A simplified C++ code to illustrate the Gauss-Jordan method is shown below. This is a modification of the code for Gauss elimination shown on page 5. Here the variable `nrhs` denotes the number of right-hand-side vectors that are used in the solution. The array `b[row,rhs]` contains all the elements of the different right-hand-side vectors. This array is also used to contain the solution. This is a common approach in linear equation solvers; the solution vector(s) are contained in the space originally used for the right-hand-side vector(s).

```
// augment a matrix with b values

for ( row = 1; row <=n; row++)
   for ( rhs = 1; rhs <=nrhs; rhs++)
      a[row][row+rhs] = b[row, rhs];

// Gauss-Jordan elimination process

for (pivot = 1; pivot <= n; pivot++ )  // do all pivot rows
{
   for ( column = pivot+1; column <= n + nrhs; column++ )
      a[pivot,column] /= a[pivot,pivot];  // make diagonal 1

   for ( row = 1; row < pivot; row++ )  // rows above pivot
      for ( column = row+1; column <= n+nrhs; column++ )
         a[row][column] -= a[row][pivot] * a[pivot][column];

   for ( row = pivot+1; row <= n; row++ )  // rows below pivot
      for ( column = row+1; column <= n+nrhs; column++ )
         a[row][column] -= a[row][pivot] * a[pivot][column];
}

// Solution complete; copy x values into original b matrix

for ( row = 1; row <= n; row++ )
   for( rhs = 1; rhs <= nrhs; rhs++ )
```

```
        b[row,rhs] = a[row,rhs+n]
```

The LU method

The LU method is a variation on the standard Gaussian elimination method. The LU method does the preliminary steps that provide a simple approach to solving a set of equations even when the right-hand-side vector, **b**, will be specified at some future time. This approach is based on converting the **A** matrix into a product of a lower triangular matrix, **L**, and an upper triangular matrix, **U**. Once these definitions are made, we can take advantage of the simple properties of triangular matrices for the easy solution of a set of equations. The LU method has the advantage that we can obtain the necessary triangular form for back-substitution even if we do not know the right-hand-side **b** vector. This is useful for doing real-time data analysis in which we receive new **b** vectors for which we want to solve the same set of simultaneous equations.

The definition of the **L** and **U** matrices is done such that

$$\mathbf{A} = \mathbf{LU} \qquad [41]$$

With the specification that **L** is lower triangular and **U** is upper triangular, we can write the matrix product in equation [41] as shown in equation [42], below.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & \cdots & a_{nn} \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & a_{n3} & \cdots & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \cdots & u_{nn} \end{bmatrix} = \mathbf{LU} \qquad [42]$$

We can see that the **U** matrix in equation [42] is just the usual upper-triangular form that results from the forward substitution part of the Gaussian elimination algorithm. This shows the similarity to Gaussian elimination.[4] The **L** matrix contains the various factors used to produce the U matrix and will be used to perform the same operations on the **b** vector.

[4] Note that both **L** and **U** have terms on the principal diagonal. The LU factorization defined here is not unique. It is possible to have the unit diagonal on the **U** matrix instead of the **L** matrix. In general, it is possible for the diagonal to have an arbitrary definition so long as it is consistent with the overall result of

Finding the coefficients in the L and U matrices

The usual formula for matrix multiplication can be applied to the **A** = **LU** product.

$$a_{ij} = \sum_{k=1}^{n} l_{ik} u_{kj} \quad [43]$$

We will use this formula to derive expressions for the components of the **L** and **U** arrays. We can do this because of the zeros in the structure of the **L** and **U** arrays. For example, the first row of the **A** matrix is the same as the first row of the **U** matrix because the only nonzero element in the first row of the **L** matrix is l_{11}, which equals one. To see this, we apply equation [43] to compute a_{1j}.

$$a_{1j} = \sum_{k=1}^{n} l_{1k} u_{kj} = l_{11}u_{1j} + (0)u_{2j} + (0)u_{3j} + \cdots + (0)\,u_{nj} = u_{1j} \quad [44]$$

Although we wrote this equation to compute a_{1j}, we already know a_{1j} from the specification of the simultaneous linear equation problem. What equation [44] gives us is the equation that we can use to compute u_{1j}.

$$u_{1j} = a_{1j} \quad [45]$$

Similarly, we can apply the general matrix multiplication equation in [43] to the first *column* of the **A** matrix, taking advantage of the zeros below the first element in the **U** matrix. This gives the following result.

$$a_{i1} = \sum_{k=1}^{n} l_{ik} u_{k1} = l_{i1}u_{11} + l_{i2}\,(0) + l_{i3}(0) + \cdots + l_{in}(0) = l_{i1}u_{11} \quad [46]$$

We can use this equation to solve for the first column of the **L** matrix, using the value of u_{11} found from equation [45].

$$l_{i1} = \frac{a_{i1}}{u_{11}} \quad [47]$$

We now have the first row of **U** and the first column of **L**. We can then proceed to compute the second row of **U**. To do this we apply equation [43] to the computation of the a_{2j} coefficients in the original matrix. Because l_{22} = 1, and all other l_{2j} with j > 2 are zero, this gives the following result.

$$a_{2j} = \sum_{k=1}^{n} l_{2k} u_{kj} = l_{21}u_{1j} + l_{22}\,u_{2j} + (0)u_{3j} + \cdots + (0)\,u_{nj} = l_{21}u_{1j} + u_{2j} \quad [48]$$

equation [39] and the restrictions that (1) **L** has no terms above the principal diagonal and (2) **U** has no terms below the principal diagonal.

We can solve this equation for u_{2j}, using the value of l_{21}, found from equation [47] (for the first column of the **L** array), and the values of $u_{1j} = a_{1j}$, found from equation [45] (for the first row of the **U** array.)

$$u_{2j} = a_{2j} - l_{21}u_{1j} \qquad j = 2,\ldots,n \tag{49}$$

Having found the second row of the **U** array, we now have sufficient information to solve for the second column of the **L** array. To do this we apply our general equation in [43] to the second column of the original **A** array.

$$a_{i2} = \sum_{k=1}^{n} l_{ik}u_{k2} = l_{i1}u_{12} + l_{i2}\,u_{22} + l_{i3}(0) + \cdots + l_{in}(0) = l_{i1}u_{12} + l_{i2}u_{22} \tag{50}$$

If we solve this equation for l_{i2}, we see that we can find all the elements from the second column of L from the original matrix coefficients a_{i2}, and previously computed values from the second column of the **U** array (u_{12} and u_{22}) and the first column of the **L** array (l_{i1}).

$$l_{i2} = \frac{a_{i2} - l_{i1}u_{12}}{u_{22}} \qquad i = 3,\ldots,n \tag{51}$$

We can generalize this approach shown here of finding one row of **U** followed by finding a column of the **L** array with the same index. The general equation for the elements of row k for the **U** matrix can be induced from equation [48]. That equation recognized that the value for l_{22}, was one and values for l_{mk}, with k > 2 were zero. To generalize equation [48] we replace the index 2 by the general row index, m, so that this becomes an equation for a_{mj}. We recognize that the diagonal term, l_{mm}, is one and values of l_{mk}, with k > m are zero. We can thus rewrite equation [48] for the general **A** matrix element, a_{mj}, as shown below.

$$a_{mj} = \sum_{k=1}^{m-1} l_{mk}u_{kj} + l_{mm}\,u_{mj} + \sum\left(from\, k = m+1\, to\, n\, is\, zero\right) \tag{52}$$

We wrote the write the $l_{mm}u_{mj}$ term, separately, because we will later set $l_{mm} = 1$. All terms in the sum for k > m are zero because values of l_{mk}, with k > m are zero. Setting l_{mm}, = 1 and solving equation [52] for u_{mj} gives the following result.

$$u_{mj} = a_{mj} - \sum_{k=1}^{m-1} l_{mk}u_{kj} \qquad j = m,\ldots,n \tag{53}$$

In a similar manner, we can generalize equation [50] to, we solve for a_{im}, instead of a_{i2}. The summation in the general equation will only run from k = 1 to k = m, because the u_{km} terms in the sum are zero for k > m. We can separate the final $l_{im}u_{mm}$ term out from the rest of the sum and solve for l_{im}, to obtain the following result for the generalization of equation [50].

$$l_{im} = \frac{a_{im} - \sum_{k=1}^{m-1} l_{ik}u_{km}}{u_{mm}} \qquad i = m+1,\ldots,n \tag{54}$$

Equations [53] and [54] provide the general approach for calculating the **L** and **U** matrix coefficients.

Computing the coefficients in the L and U matrices

In computer calculations, the **L** and **U** matrices may be stored in the same locations used for the original **A** matrix. We can see how this is possible by examining the structure of the **L** and **U** matrices shown in equation [52]. The only place that the **L** and **U** matrices overlap is on the principal diagonal. However, on this diagonal, all the diagonal elements of the **L** matrix are equal to one. If we remember that $l_{ii} = 1$ in all our calculations, we do not have to store this unit diagonal of the **L** matrix. The matrix below shows how the components of the **L** and **U** matrices (except for the l_{ii} terms on the diagonal that are all one) can be stored in the original **A** matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & \cdots & \cdots & u_{1n} \\ l_{21} & u_{22} & u_{23} & u_{24} & \cdots & \cdots & u_{2n} \\ l_{31} & l_{32} & u_{33} & u_{34} & \cdots & \cdots & u_{3n} \\ l_{41} & l_{42} & l_{43} & u_{44} & \cdots & \cdots & u_{4n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & l_{n4} & \cdots & \cdots & u_{nn} \end{bmatrix} \qquad [55]$$

The storage of **L** and **U** in the original **A** array would be done such that the storage location of a_{ij} would be used for u_{ij} if $j \geq i$; the a_{ij} storage location would be used for l_{ij} if $j < i$. This is only possible is we have no further need for a_{ij} A careful examination of equations [53] and [54] shows that once a value of a_{ij} is used in the calculation of u_{ij} or l_{ij}, it is no longer required. This we can readily store the **L** and **U** matrices in the space originally allocated for the **A** matrix.

The storage of **L** and **U** components in the space allocated for **A** makes the code simpler to write but more difficult to understand. We use the notation for the original a_{ij} matrix to refer to the components of both the **L** and **U** matrices. For example, the statements below show how we can implement equation [53] to compute u_{mj} in Fortran 95 or C++ code.

```
do j = m, n
     do k = 1, m-1
        a(m,j) = a(m,j) &
                 - a(m,k) * a(k,j)
     end do
end do
```

```
for ( j = m; j <= n; j++)
     for ( k = 1; k <= m-1; k++)
        a[m][j] -= (a[m][k] * a[k][j])
```

Anyone reviewing the code (including the original programmer) must recognize that the `a(m,j)` (or `a[m][j]`) array components in the code, sometimes refer to the components of the original **A** array, sometimes to the components of the **U** array and sometimes to the components of the **L** array. The m subscript is set in the code by an outer loop (not shown here) that does the code shown for all rows and columns. Before the loops shown above start, `a(m,j)` contains a_{mj}. At the end of the loops, u_{mj} is stored in this array component. The array components `a(m,k)` and `a(k,j)` contain l_{mk} and u_{kj}, respectively.

In the derivation of the process for finding the L/U decomposition of the **A** matrix, we outlined the calculation as proceeding by finding one row of **U** followed by finding one row of **L**. In practice we can compute the **L** and **U** arrays by looping over all columns of the **A** array. In the first column of the **A** array, we compute u_{11} and all values of l_{k1} from k = 2 to k = n. We then proceed to column 2 where we compute u_{11}, u_{22}, and all values of l_{k2}, from k = 3 to n.

The matrix below shows the formulas used for these column-by-column calculations.

$$\begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} & \cdots & \cdots & a_{1n} \\
\dfrac{a_{21}}{u_{11}} & a_{22}-l_{21}u_{12} & a_{23}-l_{21}u_{13} & a_{24}-l_{21}u_{14} & \cdots & \cdots & a_{2n}-l_{21}u_{1n} \\
\dfrac{a_{31}}{u_{11}} & \dfrac{a_{32}-l_{31}u_{12}}{u_{22}} & a_{33}-l_{31}u_{13}-l_{32}u_{23} & a_{34}-l_{31}u_{14}-l_{32}u_{24} & \cdots & \cdots & a_{3n}-l_{31}u_{1n}-l_{32}u_{2n} \\
\dfrac{a_{41}}{u_{11}} & \dfrac{a_{42}-l_{41}u_{12}}{u_{22}} & \dfrac{a_{43}-l_{41}u_{13}-l_{42}u_{23}}{u_{33}} & \begin{array}{c} a_{44}-l_{41}u_{14} \\ l_{42}u_{24}-l_{43}u_{34} \end{array} & \cdots & \cdots & \begin{array}{c} a_{4n}-l_{41}u_{1n} \\ l_{42}u_{2n}-l_{43}u_{3n} \end{array} \\
\vdots & \vdots & \vdots & \vdots & \ddots & & \vdots \\
\vdots & \vdots & \vdots & \vdots & & \ddots & \vdots \\
\dfrac{a_{n1}}{u_{11}} & \dfrac{a_{n2}-l_{n1}u_{12}}{u_{22}} & \dfrac{a_{n3}-l_{n1}u_{13}-l_{n2}u_{23}}{u_{33}} & \dfrac{a_{n4}-\sum_{k=1}^{3} l_{nk}u_{k4}}{u_{44}} & \cdots & \cdots & a_{nn}-\sum_{k=1}^{n-1} l_{nk}u_{kn}
\end{bmatrix}$$

We can convince ourselves that the column-by-column calculation works if we see that we know all the required information to calculate a given column. To do this we have to compare the calculations above with the storage pattern for the **L** and **U** matrices shown in equation [55]. The calculations shown above for the first column only require the result that $u_{11} = a_{11}$, and the data for the first column of the original **A** matrix. The calculations for the second column require **L** values from the first column, which are already known, and the values of u_{12} and u_{22}, which are found at the start of the calculations for column 2. Further examination of the columns in the matrix above shows that, for each column, the values of **L** and **U** components required in the column computations are known from previous columns. The calculation of **L** components in a column requires the values of **U** components for the same column. These will be known if we do the **U** computations for the column first.

The following C++ code will rearrange an original **A** matrix into the L/U components, stored in the same location as the original **A** matrix.

```
for ( j = 1; j <=n; j++)            // loop over all columns
{
   for ( i = 2; i <= j; i++ ) //get U components
      for(k = 1; k <=i; k++ ) a[i][j] -= a[i][k] * a[k][j];

   for ( i = j+1; i <= n; i++ ) // get L components
   {
      for ( k = 1; k < j; k++) a[i][j] -= a[i][k] * a[k][j];

      a[i][j] /= a[j][j];
   }
}
```

Actual code would have to have a strategy for ensuring that diagonal elements like u_{ii} are not zero. Additional computing strategies would look at a pivoting strategy to round-off error.

Solving the matrix equation Ax = b

Once the L/U matrix is formed, the solution of the equation **Ax** = **b** is done by defining an intermediate vector, **y**, so that the solution proceeds in two steps.

$$\begin{gathered}\mathbf{y} = \mathbf{Ux} \qquad \textit{so that } \mathbf{Ax} = \mathbf{LUx} = \mathbf{Ly} = \mathbf{b} \\ \mathbf{y} = \mathbf{L}^{-1}\mathbf{x} \qquad \textit{and} \qquad \mathbf{x} = \mathbf{U}^{-1}\mathbf{y}\end{gathered} \tag{56}$$

Solution of the equations **y** = **L**$^{-1}$**b** and **x** = **U**$^{-1}$**y** is straightforward because both **L** and **U** are triangular matrices that can be readily solved by forward and backward substitution, respectively. The equation **Ly** = **b** can be represented as follows.

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & \cdots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} \tag{57}$$

From this equation, we can see that the various values of y_i are given by the following sequence of equations.

$$\begin{gathered} y_1 = b_1 \\ y_2 = b_2 - l_{21}y_1 \\ y_3 = b_3 - l_{31}y_1 - l_{32}y_2 \end{gathered} \tag{58}$$

The general equation for this forward solution is given by equation [59].

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik}y_k \qquad i = 1,2,\ldots n \tag{59}$$

Once the **y** vector is known, the remaining matrix equation, **Ux** = **y**, has the following form.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ y_n \end{bmatrix} \qquad [60]$$

This matrix equation is solved by back substitution, starting as follows.

$$x_n = \frac{y_n}{a_{nn}}$$
$$x_{n-1} = \frac{y_{n-1} - a_{n-1\,n} x_n}{a_{n-1\,n-1}} \qquad [61]$$
$$x_{n-2} = \frac{y_{n-2} - a_{n-2\,n} x_n - a_{n-2\,n-1} x_{n-1}}{a_{n-1\,n-1}}$$

The general equation for this backward solution is given by equation [62].

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} u_{ik} x_k}{a_{ii}} \qquad i = n, n-1, n-2, \ldots, 2, 1 \qquad [62]$$

The code for the two steps in the final solution process, once the L/U components have been stored in the original **A** array, is shown below.

```
                // forward substitution for y values

for ( i = 1; i <=n; j++)
{
    y[i] = b[i];
    for ( k = 1; k < i; k++ )
        y[i] -= a[i][k] * y[k];

              // back substitution for final x values

for(i = n; i >=1; i-- )
{
    x[i] = y[i]
    for ( k = i+1; k <= n; k++)
        x[i] -= a[i][k] * x[k];

    x[i] /= a[i][i];
}
```

In a computer program, the intermediate vector, **y**, and the final solution vector, **x**, are usually stored in the same location as the original right-hand-side vector, **b**. With this storage, the code above can be replaced by the more compact code shown below.

```
                // forward substitution for y values

for ( i = 1; i <= n; j++)
{
    for ( k = 1; k < i; k++ )
        b[i] -= a[i][k] * b[k];

              // back substitution for final x values

for(i = n; i >= 1; i-- )
{
    for ( k = i+1; k <= n; k++)
        b[i] -= a[i][k] * b[k];

    b[i] /= a[i][i];
}
```

We see that all the operations on the unknowns, b_i, can be done using the information from the L/U array. It is not necessary to know the b_i array in advance, as it is with Gaussian elimination. In fact, the elements of the **L** array may are actually the multipliers that are applied to the b_i array during normal Gaussian elimination.

Pivoting strategies in solving the matrix equation Ax = b

The reduction of a matrix to its L/U form uses equations that are similar to Gaussian elimination. This reduction process is susceptible to the same problems that plague Gaussian elimination. The diagonal element, u_{mm}, that is used as a divisor in equation [54] may be zero. In addition, the arrangement of the system of equations may affect the round-off error. Both of these problems may be addressed by a strategy known as partial pivoting. In this strategy rows, but not columns, are rearranged to eliminate zeros on the diagonal and reduce round-off error.

In the partial pivoting strategy, we proceed row by row to apply equations [53] and [54]. The current row on which we are operating is called the pivot row. Before applying equation [53], we search the column with the same index as the pivot row, called the pivot column, for the maximum element (in absolute value). The row with this maximum value is then exchanged with the current pivot row. In one practice, known as scaled partial pivoting, each row is first divided by the maximum element (in absolute value) in the row. This makes the maximum element in each row equal to one.

When this exchange is done in standard Gaussian elimination both the left-hand side of the equation with the a_{ij} coefficients and the right-hand side solution vector, with components b_i, can be exchanged at the same time. In the LU method, we work only with the left-hand side coefficients. Because of this we have to create a way of remembering the row swaps that we make. This is done by creating a one dimensional integer array, `swap[row]`. Initially this array is initialized by code like the following.

```
for( int row = 1; row <=n; row++) swap[row] = row;
```

That is, the initial value of `swap[row]` is simply the row number. When two rows, say row `pivot` and row `maxPivot` are exchanged, the following code is used to keep track of the exchanges.

```
temp = swap[pivot];
```

```
            swap[pivot] = swap[maxPivot];
            swap[maxPivot] = temp;
```

When this is done consistently, the `swap[row]` array will contain the original row numbers of the equations as they are arranged at the end of the LU process. To see this, consider the example below.

Assume that the first swap occurs when row 4 is the pivot row; assume further that row 4 is to be exchanged with row 7. (pivot = 4, maxPivot = 7) Since this is the first exchange, swap(4) = 4 and swap(7) = 7 from the initialization code. The swap code then operates as follows:

```
    temp = swap[pivot];             // temp = swap[4] = 4
    swap[pivot] = swap[maxPivot];  // swap[4] = swap[7] = 7
    swap[maxPivot] = temp;          // swap[7] = temp = 4
```

We thus know that the equation whose coefficients are currently in row 7 was originally equation 4 and *vice versa.*

If there is a subsequent exchange when row 7 is the pivot row (pivot = 7), where row 11 (maxPivot = 11) is exchanged with row 7, the swap code produces the following result.

```
    temp = swap[pivot];             // temp = swap[7] = 4
    swap[pivot] = swap[maxPivot];  // swap[7] = swap[11] = 11
    swap[maxPivot] = temp;          // swap[11] = temp = 4
```

The values of the first thirteen elements in the swap(row) array, after these two exchanges, are shown in the table below.

row	1	2	3	4	5	6	7	8	9	10	11	12	13
swap[row]	1	2	3	7	5	6	11	8	9	10	4	12	13

This table shows how the `swap[row]` array tracks the changes due to row swaps. The value of the `swap[row]` array, for a particular row, is the original row number of the equation currently located in that row. For example, the value of `swap[11]` = 4 shows that the equation originally in row 4 has been moved to row 11.

The final `swap[row]` array, at the end of the formation of the LU arrays, is used to get the correct b[i] values in the back-substitution process. The structure of the LU array does not change; only the identity of the original rows has changed. Thus, when the forward-substitution process in the example here reaches the fourth row it has to use the right-hand side value from the 7th equation in the original set. The modifications to the code on pages 8 and 9 that handle the final steps with pivoting are shown below.

```
                  // forward substitution for y values

      for ( i = 1; i <= n; j++)
      {
          y[i] = b[swap[i]];
          for ( k = 1; k < i; k++ )
              y[i] -= a[i][k] * y[k];

                // back substitution for final x values

      for(i = n; i >= 1; i-- )
      {
          for ( k = i+1; k <= n; k++)
              y[i] -= a[i][k] * y[k];
```

```
        x[i]] = y[i] / a[i][i];
    }
```

In this code, the y[i] array, is used to hold intermediate values. For the ith row, this array is initialized by the statement, `y[i] = b[swap[i]]`. This places the correct right-hand-side b value into the solution process. When this code reaches the 4th row, in the example above, this statement will set y[4] = b[swap[4]] = b[7]. This statement correctly uses the right-hand side value from the 7th equation whose left-hand-side coefficients have been substituted into the 4th row. The solution process continues, however, with the correct calculations for the 4th (and subsequent) rows.

The intermediate y array is also used in the first part of the loop that obtains the final solutions. This preserves algorithm required for the back substitution process. There is no need to use the `swap[row]`array for the intermediate y[i] values or the final x[i] values. The ordering of these values depends on the column order. So long as we do not swap columns, the order of the unknowns (and the intermediate y_i variables) does not change.

The array used for the input b array may be used for storing the y values and the final x values to save storage and make the code slightly more compact (although less understandable). This assumes that there is no further need for the b array after the solution has been computed.

APPENDICES

Appendix A: Computer errors and error propagation

There are many sources of error in computer analyses. All of these have analogs to ordinary human computation. Numerical analysis schemes that represent continuous operations in calculus by approximating functions or finite difference expressions have **truncation error** that is due to the mathematical approximation. When we represent a repeating decimal or an irrational number with a fixed number of decimal places, we have a **round-off error**. Such errors are more important in computers where we can do an enormous number of sequential calculations that will expand an initial round-off error. In creating mathematical models of real engineering systems, we can have **modeling errors**. For example, the assumption of a linear stress-strain relationship may be extended into a nonlinear region. Finally, there are **blunders**. These may be errors in experimental data, incorrect data entry into a program, incorrect coding of a program, and the like.

If we perform calculations with erroneous data, we will introduce errors in our results. If two quantities, x and y, are represented by incorrect values $\tilde{x}$ and $\tilde{y}$, we define the errors in each term, ε_x, and ε_y as follows.

$$x = \tilde{x} + \varepsilon_x \qquad\qquad y = \tilde{y} + \varepsilon_y \qquad [A-1]$$

In considering error propagation, we assume that the errors in the two terms in an operation can add. Thus, for addition or subtraction we have the following result.

$$x \pm y = \tilde{x} + \varepsilon_x \pm \left(\tilde{y} + \varepsilon_y\right) = \tilde{x} \pm \tilde{y} + \left(\varepsilon_x + \varepsilon_y\right) \qquad [A-2]$$

The relative error, ε_{rel}, in this addition or subtraction operation is defined as follows.

$$\varepsilon_{rel} \equiv \frac{(x \pm y)-(\tilde{x} \pm \tilde{y})}{x \pm y} = \frac{\varepsilon_x + \varepsilon_y}{x \pm y} \approx \frac{\varepsilon_x + \varepsilon_y}{\tilde{x} \pm \tilde{y}} \qquad \text{[A-3]}$$

The final term in this equation is the estimate of the error that we can calculate. (We do not know x and y exactly, only their approximations, $\tilde{x}$ and $\tilde{y}$.) If we add two numbers with opposite signs or subtract two numbers with the same sign, we can have a very large relative error if the two numbers are close in magnitude. In some cases, such as finding the roots of a quadratic equation, it is possible to develop algorithms to avoid such large relative errors.

The relative error for multiplication is found as shown below. In this process, the product of two error terms $\varepsilon_x\varepsilon_y$ is assumed to be small compared to terms which are first order in the error.

$$xy = (\tilde{x} + \varepsilon_x)(\tilde{y} + \varepsilon_y) = \tilde{x}\tilde{y} + (\tilde{y}\varepsilon_x + \tilde{x}\varepsilon_y) + \varepsilon_x\varepsilon_y \approx \tilde{x}\tilde{y} + (\tilde{y}\varepsilon_x + \tilde{x}\varepsilon_y) \qquad \text{[A-4]}$$

$$\varepsilon_{rel} = \frac{xy - \tilde{x}\tilde{y}}{xy} \approx \frac{\tilde{y}\varepsilon_x + \tilde{x}\varepsilon_y}{xy} \approx \frac{\tilde{y}\varepsilon_x + \tilde{x}\varepsilon_y}{\tilde{x}\tilde{y}} = \frac{\varepsilon_x}{\tilde{x}} + \frac{\varepsilon_y}{\tilde{y}} \qquad \text{[A-5]}$$

The derivation of the relative error for division, shown below, assumes that the relative error in the divisor, measured as $\varepsilon_y/\tilde{y}$, is small to allow the use of a series expansion for 1/(1+x) which only converges if |x| is less than one.

$$\frac{x}{y} = \frac{\tilde{x} + \varepsilon_x}{\tilde{y} + \varepsilon_y} = \frac{\dfrac{\tilde{x}}{\tilde{y}} + \dfrac{\varepsilon_x}{\tilde{y}}}{1 + \dfrac{\varepsilon_y}{\tilde{y}}} = \left(\frac{\tilde{x}}{\tilde{y}} + \frac{\varepsilon_x}{\tilde{y}}\right)\left[1 - \frac{\varepsilon_y}{\tilde{y}} - \left(\frac{\varepsilon_y}{\tilde{y}}\right)^2 + \cdots\right] \approx \frac{\tilde{x}}{\tilde{y}} + \frac{\varepsilon_x}{\tilde{y}} - \frac{\tilde{x}}{\tilde{y}}\frac{\varepsilon_y}{\tilde{y}} \qquad \text{[A-6]}$$

As usual, we assume that the errors add so that we have the following result for error propagation in division.

$$\varepsilon_{rel} = \frac{\dfrac{x}{y} - \dfrac{\tilde{x}}{\tilde{y}}}{\dfrac{x}{y}} \approx \frac{\dfrac{\varepsilon_x}{\tilde{y}} + \dfrac{\tilde{x}}{\tilde{y}}\dfrac{\varepsilon_y}{\tilde{y}}}{\dfrac{x}{y}} = \frac{\dfrac{\tilde{x}}{\tilde{x}}\dfrac{\varepsilon_x}{\tilde{y}} + \dfrac{\tilde{x}}{\tilde{y}}\dfrac{\varepsilon_y}{\tilde{y}}}{\dfrac{x}{y}} = \frac{\dfrac{\tilde{x}}{\tilde{y}}\left(\dfrac{\varepsilon_x}{\tilde{x}} + \dfrac{\varepsilon_y}{\tilde{y}}\right)}{\dfrac{x}{y}} \approx \frac{\varepsilon_x}{\tilde{x}} + \frac{\varepsilon_y}{\tilde{y}} \qquad \text{[A-7]}$$

Thus, both multiplication and have the same result for the combined error.

The derivations above apply to individual operations and look at the worst-case condition where the errors are assumed to add. Advanced analyses of error propagation take a statistical approach to the analysis of error propagation and produce less severe errors. However, the size of the errors in a set of calculations increases as the number of required calculations increases.

Ill-conditioned problems – Certain problems led to difficulties in solution because small changes in the inputs can make a large change in the results. As an example of this, consider the following algorithm known as Newton's method for the solution of one equation in one unknown. The equation is expressed as f(x) = 0 and we seek the value of x that makes this true. We denote the successive guesses to the solution as x_0, x_1, …x_n, … . The general rule for going from one iteration on x, x_n to a new iteration value, x_{n+1}, is given by the following equation.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad \text{[A-8]}$$

We assume that f(x) is a function whose first derivative, f`(x) = df/dx, can be computed.

Newton's method can be expanded to a system of nonlinear equations. We assume that we have N equations in N unknowns. Each individual equation can be written in the form $f_i(x_1, x_2, \ldots x_N) = 0$ where we have N such equations from $f_1 = 0$ to $f_N = 0$. We can represent the set of equations in the vector notation $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. Our iterations in this multidimensional system take us from one multidimensional point at iteration n, $\mathbf{x}^{(n)}$, to the new iteration point, $\mathbf{x}^{(n+1)}$. To get this iteration equation for this process we expand each equation in a Taylor series about fixed point in our N-dimensional space, $\mathbf{x}^{(n)}$. If we drop terms that are second order and higher this series becomes.

$$\mathbf{f}\left(\mathbf{x}^{(n+1)}\right) = \mathbf{f}\left(\mathbf{x}^{(n)}\right) + \mathbf{J}^{(n)}\left(\mathbf{x}^{(n+1)} - \mathbf{x}^{(n)}\right) + \cdots \qquad \text{[A-9]}$$

In this series, we have defined $\mathbf{J}^{(n)}$ as the matrix of partial derivatives: $\mathbf{J}^{(n)} = \partial f_k / \partial x_m \big|^{(n)}$, and the (n) superscript in this definition indicates that the partial derivatives are evaluated at the point $\mathbf{x} = \mathbf{x}^{(n)}$. Writing equation [A-9] in full matrix form, with higher-order terms neglected, gives the following result.

$$\begin{bmatrix} f_1(\mathbf{x}^{(n+1)}) \\ f_2(\mathbf{x}^{(n+1)}) \\ f_3(\mathbf{x}^{(n+1)}) \\ \vdots \\ \vdots \\ f_N(\mathbf{x}^{(n+1)}) \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{x}^{(n)}) \\ f_2(\mathbf{x}^{(n)}) \\ f_3(\mathbf{x}^{(n)}) \\ \vdots \\ \vdots \\ f_N(\mathbf{x}^{(n)}) \end{bmatrix} + \begin{bmatrix} J_{11} & J_{12} & J_{13} & \cdots & \cdots & J_{1N} \\ J_{21} & J_{22} & J_{23} & \cdots & \cdots & J_{2N} \\ J_{31} & J_{32} & J_{33} & \cdots & \cdots & J_{3N} \\ \vdots & \vdots & \vdots & \ddots & & \\ \vdots & \vdots & \vdots & & \ddots & \\ J_{N1} & J_{N2} & J_{N3} & \cdots & \cdots & J_{NN} \end{bmatrix} \begin{bmatrix} x_1^{(n+1)} - x_1^{(n)} \\ x_2^{(n+1)} - x_2^{(n)} \\ x_3^{(n+1)} - x_3^{(n)} \\ \vdots \\ \vdots \\ x_N^{(n+1)} - x_N^{(n)} \end{bmatrix} \qquad \text{[A-10]}$$

We can also write equation [A-10] in a summation form for the general function, f_k.

$$f_k(\mathbf{x}^{(n+1)}) = f_k(\mathbf{x}^{(n)}) + \sum_{m=1}^{N} J_{km}\left(x_m^{(n+1)} - x_m^{(n)}\right) = f_k(\mathbf{x}^{(n)}) + \sum_{m=1}^{N} \frac{\partial f_k}{\partial x_m}\bigg|^{(n)} \left(x_m^{(n+1)} - x_m^{(n)}\right)$$

[A-11]

Since the solution we seek is for $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, we set $f_k(\mathbf{x}^{(n+1)}) = 0$ for k = 1, …, N. Since the functions are, in general, nonlinear, we do not expect this to give us the correct solution. However, we hope that it will provide an iterative procedure that will eventually lead to a solution. Setting the left-hand side of equation [A-11] equal to zero gives us the following system of linear equations to solve for the increment in **x**.

$$\sum_{m=1}^{N} J_{km}\left(x_m^{(n+1)} - x_m^{(n)}\right) = \sum_{m=1}^{n} \left.\frac{\partial f_k}{\partial x_m}\right|^{(n)} \left(x_m^{(n+1)} - x_m^{(n)}\right) = -f_k(\mathbf{x}^{(n)}) \quad k = 1, n \quad \text{[A-12]}$$

At each iteration step in we have to solve the system of linear equations in [A-12] to obtain the new values for our unknowns, $x_m^{(n+1)}$.

An example of ill conditioning is this process occurs in solving thermodynamic properties for liquids. Thermodynamic property equations for real substances are usually written in terms of temperature and specific volume (or temperature and density) as independent variables. If we are trying to find the temperature and density that give a specified entropy and pressure, we are trying to solve equations of the form $s – s_0 = 0$ and $P – P_0 = 0$. In this case equation [A-12] represents the two following equations.

$$\begin{aligned}
\frac{\partial s}{\partial T}\left(T^{(n+1)} - T^{(n)}\right) + \frac{\partial s}{\partial v}\left(v^{(n+1)} - v^{(n)}\right) &= s_0 - s\left(T^{(n)}, v^{(n)}\right) \\
\frac{\partial P}{\partial T}\left(T^{(n+1)} - T^{(n)}\right) + \frac{\partial P}{\partial v}\left(v^{(n+1)} - v^{(n)}\right) &= P_0 - P\left(T^{(n)}, v^{(n)}\right)
\end{aligned} \quad \text{[A-13]}$$

We can apply Cramer's rule to write the solution for the changes in temperature and specific volume (at each iteration) as follows.

$$\begin{aligned}
\left(T^{(n+1)} - T^{(n)}\right) &= \frac{\frac{\partial P}{\partial v}\left[s_0 - s\left(T^{(n)}, v^{(n)}\right)\right] - \frac{\partial s}{\partial v}\left[P_0 - P\left(T^{(n)}, v^{(n)}\right)\right]}{\frac{\partial s}{\partial T}\frac{\partial P}{\partial v} - \frac{\partial s}{\partial v}\frac{\partial P}{\partial T}} \\
\left(v^{(n+1)} - v^{(n)}\right) &= \frac{\frac{\partial s}{\partial T}\left[P_0 - P\left(T^{(n)}, v^{(n)}\right)\right] - \frac{\partial P}{\partial T}\left[s_0 - s\left(T^{(n)}, v^{(n)}\right)\right]}{\frac{\partial s}{\partial T}\frac{\partial P}{\partial v} - \frac{\partial s}{\partial v}\frac{\partial P}{\partial T}}
\end{aligned} \quad \text{[A-14]}$$

The partial derivative, $\partial P/\partial v$, is infinity for an incompressible fluid. For liquids, which are virtually incompressible, this is a very large number. In the volume correction equation, this derivative appears in both the numerator and denominator. However, in the temperature correction equation it appears in the denominator only. In both equations the terms in the numerator, $s_0 – s(T,v)$ and $P_0 – P(T,v)$, represent the difference between the desired entropy and pressure values the current guesses for those quantities. These differences in the temperature correction equation are divided by the large value of $\partial P/\partial v$, which dominates the denominator. Because of this the temperature correction can easily become zero to within the machine epsilon even where there are still significant differences between the guesses and desired values for pressure and entropy.

Appendix B: Effect of changes in A and b in the solution of Ax = b

If the data in the **A** matrix are changed by an amount **δA** (or if there are errors of this amount), the solution will change by an amount **δx**, such that

$$(\mathbf{A}+\delta\mathbf{A})(\mathbf{x}+\delta\mathbf{x})=\mathbf{b} \quad \Rightarrow \quad \mathbf{Ax}+\mathbf{A}\,\delta\mathbf{x}+\delta\mathbf{Ax}+\delta\mathbf{A}\delta\mathbf{x}=\mathbf{b} \qquad \text{[B-1]}$$

Subtracting **Ax** = **b** from equation [B-1] gives

$$\mathbf{A}\,\delta\mathbf{x}+\delta\mathbf{Ax}+\delta\mathbf{A}\delta\mathbf{x}=\mathbf{0} \quad \Rightarrow \quad \mathbf{A}\,\delta\mathbf{x}=-\delta\mathbf{Ax}+\delta\mathbf{A}\delta\mathbf{x} \qquad \text{[B-2]}$$

Premultiplying the second equation in [B-2] by $\mathbf{A}^{-1}$ gives (after noting that $\mathbf{A}^{-1}\mathbf{A}\delta\mathbf{x} = \mathbf{I}\delta\mathbf{x} = \delta\mathbf{x}$)

$$\delta\mathbf{x}=-\mathbf{A}^{-1}\delta\mathbf{Ax}+\mathbf{A}^{-1}\delta\mathbf{A}\delta\mathbf{x}=-\mathbf{A}^{-1}\delta\mathbf{A}(\mathbf{x}+\delta\mathbf{x}) \qquad \text{[B-3]}$$

Taking norms of equation [B-3] and applying the inequality that $||\mathbf{Ax}|| \le ||\mathbf{A}||\,||\mathbf{x}||$ two times gives the result in [B-4]. (Notice that taking the norm removes the minus sign, since the norm, as a measure of size, is a positive quantity. This is also true in mechanics. A vector velocity of -10 m/s has a magnitude of 10 m/s.)

$$\|\delta\mathbf{x}\|=\left\|\mathbf{A}^{-1}\delta\mathbf{A}(\mathbf{x}+\delta\mathbf{x})\right\|\le\left\|\mathbf{A}^{-1}\right\|\,\left\|\delta\mathbf{A}(\mathbf{x}+\delta\mathbf{x})\right\|\le\left\|\mathbf{A}^{-1}\right\|\,\|\delta\mathbf{A}\|\|\mathbf{x}+\delta\mathbf{x}\| \qquad \text{[B-4]}$$

We can divide equation [B-4] by $||\mathbf{x} + \mathbf{dx}||$ use the condition number κ(**A**), defined in equation [35] as the product of the norms, $||\mathbf{A}||\,||\mathbf{A}^{-1}||$ to replace $||\mathbf{A}^{-1}||$.

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}+\delta\mathbf{x}\|}\le\left\|\mathbf{A}^{-1}\right\|\,\|\delta\mathbf{A}\|=\frac{\kappa(\mathbf{A})}{\|\mathbf{A}\|}\|\delta\mathbf{A}\|=\kappa(\mathbf{A})\frac{\|\delta\mathbf{A}\|}{\|\mathbf{A}\|} \qquad \text{[B-5]}$$

Equation [B-5] tells us that the relative change in the norm of the solution is less than the relative change in the norm of the **A** matrix times the condition number, κ(**A**). A large condition number will amplify the effect of changes (or data errors) in the **A** matrix on the solution.

Similarly, if the data in the **b** vector are changed by an amount **δb** (or if there are errors of this amount), the solution will change by an amount **δx**, such that

$$\mathbf{A}(\mathbf{x}+\delta\mathbf{x})=\mathbf{b}+\delta\mathbf{b} \quad \Rightarrow \quad \mathbf{Ax}+\mathbf{A}\,\delta\mathbf{x}=\mathbf{b}+\delta\mathbf{b} \qquad \text{[B-6]}$$

Subtracting **Ax** = **b** from equation [B-6] gives

$$\mathbf{A}\,\delta\mathbf{x}=\delta\mathbf{b} \qquad \text{[B-7]}$$

Premultiplying equation [B-7] by $\mathbf{A}^{-1}$ gives (after noting that $\mathbf{A}^{-1}\mathbf{A}\delta\mathbf{x} = \mathbf{I}\delta\mathbf{x} = \delta\mathbf{x}$)

$$\delta\mathbf{x}=-\mathbf{A}^{-1}\delta\mathbf{b} \qquad \text{[B-8]}$$

Taking norms of equation [B-8], applying the inequality that $||\mathbf{Ax}|| \le ||\mathbf{A}||\,||\mathbf{x}||$, and replacing the norm $||\mathbf{A}^{-1}||$ by the ratio κ(**A**)/||**A**|| gives the result in [B-9].

$$\|\boldsymbol{\delta}\mathbf{x}\| = \|\mathbf{A}^{-1}\boldsymbol{\delta}\mathbf{b}\| \le \|\mathbf{A}^{-1}\|\,\|\boldsymbol{\delta}\mathbf{b}\| = \kappa(\mathbf{A})\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{A}\|} \qquad \text{[B-9]}$$

We can divide the inequality [B-9] by **||x||** to obtain

$$\frac{\|\boldsymbol{\delta}\mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(\mathbf{A})\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{A}\|\,\|\mathbf{x}\|} \qquad \text{[B-10]}$$

Since **Ax** = **b**, we have the following relationship for norms

$$\|\mathbf{b}\| = \|\mathbf{A}\mathbf{x}\| \le \|\mathbf{A}\|\,\|\mathbf{x}\| \qquad \text{[B-11]}$$

If we replace **||A||** **||x||** in the denominator of the larger term in the inequality [B-10] by **||b||**, we will be dividing by a smaller term. Thus, the inequality will still hold. This leads to our final conclusion shown below.

$$\frac{\|\boldsymbol{\delta}\mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(\mathbf{A})\frac{\|\boldsymbol{\delta}\mathbf{b}\|}{\|\mathbf{b}\|} \qquad \text{[B-12]}$$

Again, we see that the condition number of the matrix determines the relationship between the relative changes in data (or errors) in the **b** vector and the relative changes in the solution, **x**.

Appendix C: Notes on binary numbers

Introduction – Most people have heard that a computer is a binary device, but many students are not familiar with the details of binary numbers and their representation on a computer. This appendix provides background on binary numbers and their use in computer representation of numerical data. Typical representations in C++ are used as examples, but the representation and some of the quantitative limits are similar for other languages.

Binary numbers – The interpretation of binary numbers is a specific application of the general approach to representing numbers in any base. In our usual base ten, we recognize that the number 132 represents 1 times 10^2 plus 3 times 10^1 + 2 times 10^0. We can create a formula for representing this number in base ten and then generalize if to other bases. To start with we write the three digits in our number (the 1, the 3, and the 2 in the number 132) as the symbol, d_i, for the digit in position i. We number the digits from right to left, starting with the symbol d_0 for the rightmost digit, which is 2 in this example of the number 132. We then say that the first digit to the left of d_0 is d_1, which is 3 in this example. The next digit to the left is d_2, which is 1 in our example. We can represent a general three-digit number by the symbols $d_2d_1d_0$. Since this is a base-ten number, we can calculate the value of the number as $d_2(10^2) + d_1(10^1) + d_0(10^0)$. Note that the range of digits for a base ten number is zero to nine.

How can we make this more general? We want to represent a number, in any base, with an arbitrary amount of digits in the number. For a base-ten number with N digits, we can write the following equation for the value of the number.

$$d_{N-1}d_{N-2}\ldots\ldots d_1d_0 = d_{N-1}(10^{N-1}) + d_{N-2}(10^{N-2})\ldots\ldots + d_1(10^1) + d_{01}(10^0)$$

Here, we have written the number on the left as a sequence of digits and on the right as the calculation, which gives its value. Note that we number the digits from zero to N-1 for our total of N digits. We can rewrite the equation above using the usual sigma notation for sums.

$$d_{N-1}d_{N-2}\ldots\ldots d_1 d_0 = \sum_{k=0}^{N-1} d_k (10^k)$$

In this equation, we use ten because that is the base of our usual number system. To consider other number systems such a binary (base 2), octal (base 8), or hexadecimal (base 16), we use the same formula, but we replace the factor to ten by the appropriate base, b.

$$d_{N-1}d_{N-2}\ldots\ldots d_1 d_0 = \sum_{k=0}^{N-1} d_k (b^k)$$

When want to formally represent the base that we are using for the number, we include it as a subscript following the number. Thus, the base-ten number shown above as 132 could be formally written as 132_{10} to emphasize that this is a base-ten number.

Note that the value of any "digit", d_k in a given numbering system must be less than the base b. We know that decimal (base ten) digits range from zero to 9; similarly, octal digits range from 0 to 7 and binary digits range from 0 to 1. Hexadecimal digits range from 0 to f, where we have the following correspondence between hexadecimal digits and base ten numbers: $a_{16} = 10_{10}$; $b_{16} = 11_{10}$; $c_{16} = 12_{10}$; $d_{16} = 13_{10}$; $e_{16} = 14_{10}$; $f_{16} = 15_{10}$. We can write our example number 132_{10} in the following bases: $132_{10} = 10000100_2 = 204_8 = 84_{16}$. The table below shows the numbers in four bases (decimal, binary, octal and hexadecimal) for numbers from 0_{10} to 19_{10}.

Numbers from 0_{10} to 19_{10} in Four Bases

Decimal	0	1	2	3	4	5	6	7	8	9
Binary	0	1	10	11	100	101	110	111	1000	1001
Octal	0	1	2	3	4	5	6	7	10	11
Hex	0	1	2	3	4	5	6	7	8	9
Decimal	10	11	12	13	14	15	16	17	18	19
Binary	1010	1011	1100	1101	1110	1111	10000	10001	10010	10011
Octal	12	13	14	15	16	17	20	21	22	23
Hex	a	b	c	d	e	f	10	11	12	13

For binary numbers the only digits we can have are zero and one. Thus, our general-base equation is written as follows for binary (b = 2) numbers.

$$d_{N-1}d_{N-2}\ldots\ldots d_1 d_0 = \sum_{k=0}^{N-1} d_k (2^k)$$

How large a number can we represent on a computer if we use 16 binary digits? (A binary digit is usually called a bit and eight binary digits or bits are called a byte.) The minimum value of our 16-bit binary number is zero and the maximum occurs when all the bits are one. With N = 16, our sum ranges from 0 to 15 and we get the following result when all sixteen bits are one.

$$d_{15}d_{14}\ldots\ldots d_1 d_0 = \sum_{k=0}^{15} d_k(2^k) \qquad so\,that$$

$$1111111111111111 = \sum_{k=0}^{15} 1(2^k) = 2^{15} + 2^{14} + \ldots\ldots + 2^1 + 2^0 = 65\,535$$

In any case like this one, where all the binary bits are one, we can add one to the number to obtain the following result.

$$1_{N-1}1_{N-2}\ldots\ldots 1_1 1_0 + 1 = \sum_{k=0}^{N-1} 2^k + 1 = 1_N 0_{N-1} 0_{N-2}\ldots\ldots 0_1 0_0 = 2^N + \sum_{k=0}^{N-1} 0(2^k) = 2^N$$

$$so\ that \quad 1_{N-1}1_{N-2}\ldots\ldots 1_1 1_0 = 2^N - 1$$

We can express this result as follows: **the maximum value of a binary number with N digits is 2^N – 1.** If N = 16, we obtain the result shown previously: the maximum number that we can represent is 2^{16} – 1 = 65 535$_{10}$. If we wanted to use these 16 bits to represent both positive and negative numbers, we could interpret the electronic bits in the computer so that the electronic binary 0 was the lowest number and the computer value of 1000000000000000$_2$ = 32 768$_{10}$ would represent zero. In this case our computer circuit would have to subtract 32 768$_{10}$ from the binary bits stored in memory to give us our actual number. Since our 16-bit number ranges from 0 to 65 535$_{10}$ in the computer memory, subtracting 32 768$_{10}$ from this range gives us a range of -32 768$_{10}$ to 32 767$_{10}$ for our 16-bit number when we want to have negative numbers.

Of course, choosing more or less than 16 bits would affect the range of our number. In the next section, we discuss the various data types and show how their allowed ranges are governed by the way in which the numbers are represented as binary information.

Visual C++ as an example of typical data types – C++ has two basic kinds of numerical data types: integer numbers without decimal points and “floating point” numbers which have a decimal point. (Some accounting systems have “fixed point” systems that have a fixed range for decimal numbers. Some specialized programs like MATLAB have variable precision where a user can specify the number of significant digits.) The C++ standard allows the size of integer data types to be set by the compiler vendors, but there are a set of rules that govern the relationship among the sizes of various data types. The ***names*** of the integer data types shown below are common to all C++ implementations. The ***ranges*** shown for each type are those used in Visual C++. However, these ranges are common for many other compilers. (Note that the int data type, according to the C++ standard, can have a range anywhere between the short and the long):

short	a 16-bit integer data type with range –32 768 to 32 767
int	a 32-bit integer data type with range –2 147 483 648 to 2 147 483 647
long	a 32-bit integer data type with range –2 147 483 648 to 2 147 483 647
unsigned short	a 16-bit integer data type with range 0 to 65,535
unsigned int	a 32-bit integer data type with range 0 to 4 294 967 295
unsigned long	a 32-bit integer data type with range 0 to 4 294 967 295

The range for the short and the unsigned short are just the numbers we that we found above when analyzing the 16-bit binary number. For the 32-bit number, the maximum range is from 0 to 2^{32} -1 = 4 294 967 295. This is the range for the unsigned long and unsigned int. When we want negative integers, we use half this range for negative numbers and half for positive numbers and zero.

Floating point data types are represented in the binary equivalent of a power notation. That is we can represent numbers like 6.02x10^{23} and 1.38x10^{-16} in terms of their power of ten (called the characteristic) and their prefactor (called the mantissa). We also require a bit to tell us if the number is positive or negative. The formats of floating point numbers used in computer are governed by an international standard that started as IEEE standard 754 in 1985. In this standard a double precision floating point number is stored in eight bytes or 64 bits. These 64 bits are used as follows: 1 sign bit, 11 bits for the (binary) mantissa, and 52 bits for the characteristic.

With 11 bits for the mantissa, we can handle a range from 0 to $2^{11} – 1 = 2047$. The maximum value in this range is $2^{2047} \approx 10^{616}$. This range is usually divided into a range between 10^{-308} and 10^{308}. The characteristic will always start with a 1. This 1 is not stored so the effective number of bits for the mantissa is really 52 +1 = 53. Fifty-three binary bits can store a number up to 2^{53}-1 = 9.007x10^{15}. Thus the characteristic can store almost sixteen significant decimal digits.

C++ has three floating point types **float**, **double**, and **long double**. Each of these data types represents a limited range of positive and negative numbers and zero. The range for float is shown below.

-3.402823466x10^{38} to -1.175494351x10^{-38}, 0, and 1.175494351x10^{-38} to -3.402823466x10^{38}

The range for type double is shown next:

-1.7976931348623158x10^{308} to -2.2250738585072014x10^{-308}, 0, and

2.2250738585072014x10^{-308} to 1.7976931348623158x10^{308}

In Microsoft Visual C++, there is no difference between a double and a long double. Other compiler vendors, who do offer a difference, usually have has a range of approximately 3.4x10^{-4932} to 3.4x10^{-4932}.

The float data type occupies 4 bytes or 32 bits; the double type occupies 8 bytes or 64 bits; the long double described in the previous paragraph occupies 10 bytes or 80 bits. The number of bits for the mantissa defines the exponent range and the number of bits for the exponent defines the number of significant figures. The float type represents about seven significant figures exactly. Double and long double (in systems where long double uses more than 8 bytes for storage) represent about 15 and 19 significant figures, respectively.

Integer data types are usually handled on the computer so that there is no check on the maximum or minimum value. Because of this you can do something like the following.

```
short x, y = 32767;     // sets y to maximum int

x = y + 1;              // can we exceed the maximum?

cout << x;              // surprise!  x = -32768!
```

Adding one to the maximum integer gives the minimum integer. For floating point variables exceeding the maximum or going below the minimum gives you an overflow or underflow error, respectively. These numbers are represented on the computer as 1.#INF and -1.#INF, respectively. Another number you may see displayed on your computer screen is -1,#IND; the IND is an abbreviation for INDefinite; this results from attempted mathematical operations such as dividing zero by zero, raising zero to the zeroth power, etc.