Section 3 - Systems of Linear Algebraic Equations

In this section, we will explore the solution of systems of equations of the type **Ax** = **B**, where **A** is the matrix of coefficients of the system of equations, is a vector of unknown variables and **B** is a vector of known quantities, the "right-hand-sides" of the system of equations.

With equations of this type, three possibilities exist: the system can have a unique solution (**A**≠0), it can have no solution (**A**=0, **B**≠0), or it can have infinitely many (non-unique) solutions (**A**=0, **B**=0). One of the tasks we will always have is to identify which of these cases we are dealing with as we attempt to solve the problem.

3.1 Elimination Methods

3.1.1 Gauss Elimination

Consider the following system of equations:

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1 \\
&a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = b_2 \\
&a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n = b_3 \\
&\ldots \\
&a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n = b_n
\end{aligned}$$

Multiply the first row (the *pivot row*) by a_{21}/a_{11} and subtract it from the second row, multiply the first equation by a_{31}/a_{11} and subtract it from the third row, etc., to yield

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1 \\
&0 + a'_{22}x_2 + a'_{23}x_3 + \ldots + a'_{2n}x_n = b'_2 \\
&0 + a'_{32}x_2 + a'_{33}x_3 + \ldots + a'_{3n}x_n = b'_3 \\
&\ldots \\
&0 + a'_{n2}x_2 + a'_{n3}x_3 + \ldots + a'_{nn}x_n = b'_n
\end{aligned}$$

The second row now becomes the pivot row and the method proceeds (*forward elimination*) until only one known remains and **A** is an upper triangular matrix:

$$\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1 \\
&0 + a'_{22}x_2 + a'_{23}x_3 + \ldots + a'_{2n}x_n = b'_2 \\
&\ldots \\
&0 + 0 + \ldots + a^{(n-1)}_{nn}x_n = b^{(n-1)}_n
\end{aligned}$$

To complete the solution of the system of equations, *back-substitute* to find

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

$$x_{n-1} = \frac{b_{n-1}^{(n-2)} - a_{(n-1)n}^{(n-2)} x_n}{a_{(n-1)(n-1)}^{(n-2)}}$$

etc.

Operation count - Adding up the number of floating point operations (*flops*) gives $2n^3/3 + O(n^2)$ for the forward elimination and $n^2 + O(n)$ for the back substitution, and $2n^3/3 + O(n^2)$ for the entire naïve method. Gauss Elimination, therefore, gets very costly as n increases, and most of the effort (and time) is in the elimination phase of the method.

Improvements to, and considerations for, the naïve Gauss Elimination method:

- Division by zero – Must avoid zeros as the diagonal entries of the pivot rows.
- Round-off errors – Always check the final solution by substituting it into the original equations, although this is not always a good measure of the validity of the solution with poorly-conditioned systems.
- Ill-conditioned systems – Systems that produce large changes in results with small changes in inputs, characterized by small determinants or nearly singular systems.
- Scaling – Multiplying entire equations (rows) by constants to reduce subtractive cancellation (rarely done in practice).
- Pivoting – Always done in practice. Reorder the equations beginning with the pivot row so that the largest coefficient in that column is the one that is pivoted (*partial pivoting*). Pivoting can also be done with columns (*full pivoting*) to result in a diagonal matrix, but since it changes the order of the variables it is rarely done in practice. Rarely does a commercial code actually interchange the rows since this takes time; instead an index vector keeps track of the location of the rows.

Pseudocode – Gauss Elimination with partial pivoting and pivot scaling

```
SUB Gauss(a, b, n, x, tol, er)
      DIMENSION s(n) ' largest element in each row, for scaling
      er = 0
      DOFOR i = 1, n
            s(i) = ABS(A(i,1))
            DOFOR j = 2, n
                  IF ABS(a(i,j)) > s(i) THEN s(i) = ABS(a(i,j))
            END DO
      END DO
      CALL Eliminate(a, s, n, b, tol, er)
      IF er<> -1 THEN
            CALL Substitute(a, n, b, x)
      END IF
END Gauss

SUB Eliminate(a, s, n, b, tol, er)
      DOFOR k = 1, n-1
            CALL Pivot(a, b, s, n, k)
```

```
            IF ABS(a(k,k)/s(k))<tol THEN
                  er=-1
                  EXIT DO
            END IF
            DOFOR i = k+1, n
                  factor = a(i,k)/a(k,k)
                  DOFOR j = k+1, n
                        a(i,j) = a(i,j) - factor*a(k,j)
                  END DO
                  b(i) = b(i) - factor*b(k)
            END DO
      END DO
      IF ABS(a(n,n)/s(n))<tol THEN er=-1
END Eliminate

SUB Pivot(a, b, s, n, k)
      p = k
      big = ABS(a(k,k)/s(k))
      DOFOR ii = k+1, n
            dummy = ABS(a(ii,k)/s(ii))
            IF dummy > big THEN
                  big = dummy
                  p = ii
            END IF
      END DO
      IF p <> k THEN
            DOFOR jj = k, n ' swap the rows
                  dummy = a(p,jj)
                  a(p,jj) = a(k,jj)
                  a(k,jj) = dummy
            END DO
            dummy = b(p) ' swap the RHS
            b(p) = b(k)
            b(k) = dummy
            dummy = s(p) ' swap the scale factors
            s(p) = s(k)
            s(k) = dummy
      END IF
END Pivot

SUB Substitute(a, n, b, x)
      x(n) = b(n) / a(n,n)
      DOFOR i = n-1, 1, -1
            sum = 0
            DOFOR j = i+1, n
                  sum = sum + a(i,j)*x(j)
            END DO
            x(i) = (b(i) - sum) / a(i,i)
      END DO
END Substitute
```

3.1.2 Gauss-Jordan

The Gauss-Jordan method is a variation of Gauss Elimination where all of the matrix rows are reduced by the current row, not just the subsequent ones. In addition, the current row is

normalized with respect to its diagonal element. This results in the identity matrix and the right-hand-side is equal to the solution vector after the method is complete. However, the operation count is significantly higher than with Gauss Elimination (about 50% higher due to the larger number of elimination steps), so it is rarely used in practice.

3.2 LU Decomposition and Matrix Inversion

Gauss elimination is designed to solve equations of the type **Ax** = **B**. Applying this method as described above becomes inefficient if there are several vectors **B** that must be evaluated. Recall that Gauss Elimination consists of forward elimination and back substitution, and that the larger effort is in the forward elimination.

LU decomposition methods separate the elimination step from the back substitution, by "decomposing" **A** into an upper triangular matrix (**U**) and a lower triangular matrix (**L**), so that

	$\mathbf{Ax - B = 0 \rightarrow UX - D = 0}$
Premultiply by **L:**	$\mathbf{LUX - LD = Ax - B}$
Therefore	$\mathbf{LU = A}$, $\mathbf{LD = B}$

There are two distinct steps in LU decomposition: The first is the decomposition step where **A** is factored or *decomposed* into lower (**L**) and upper (**U**) triangular matrices. The second is the substitution step where **L** and **U** are used to determine a solution for a particular right-hand-side vector **B** by generating an intermediate vector **D** by forward substitution and then back substituting to find **x**.

3.2.1 LU Decomposition using Gauss Elimination

Recall that Gauss Elimination results in an upper triangular matrix of the form

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix}$$

There is also a lower matrix developed in the process. Recall that we multiplied the first row by $f_{21} = a_{21}/a_{11}$ and subtracted it from the second row, multiplied the first row by $f_{31} = a_{31}/a_{11}$ and subtracted it from the third row. Finally, we multiply the second row by $f_{32} = a'_{32}/a'_{22}$ and subtract it from the third row. We don't have to do these operations on the right-hand-side **B**, we can save the values to manipulate it later. Store the numbers in the decomposed matrix **A**:

$$\mathrm{A = LU} = \begin{bmatrix} 1 & 0 & 0 \\ f_{21} & 1 & 0 \\ f_{31} & f_{32} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix}$$

To evaluate the vector corresponding to **B**, forward substitute to find the elements of **D**:

$$d_i = b_i - \sum_{j=1}^{i-1} a_{ij} d_j \quad , i = 2, 3, \ldots n$$

then back substitute to find

$$x_n = \frac{d_n}{a_{nn}}$$

$$x_i = \frac{d_i - \sum_{j=i+1}^{n} a_{ij} x_j}{a_{ii}} \quad , i = n\text{-}1, n\text{-}2, ..., 1$$

This method has the same number of operations as Gauss Elimination for one right-hand-side vector **B**. If more than one right-hand-side vector **B** is to be evaluated, this LU decomposition method (referred to as *Doolittle decomposition*) is more efficient since it saves the effort and time of the forward elimination.

Pseudocode – LU Decomposition using Gauss-Elimination, partial pivoting with order vector

```
SUB LU_Decomp(a, b, n, x, tol, er)
      DIMENSION s(n) ' largest element in each row, for pivot scaling
      DIMENSION o(n) ' row order vector
      er = 0
      CALL LU_Decompose(a, n, tol, o, s, er)
      IF er <> -1 THEN
            CALL LU_Substitute(a,o,n,b,x)
      END IF
END LU_Decomp

SUB LU_Decompose(a, n, tol, o, s, er)
      DOFOR i = 1, n
            o(i) = i
            s(i) = ABS(a(i,1))
            DOFOR j = 2, n
                  IF ABS(a(i,j)) > s(i) THEN s(i) = ABS(a(i,j))
            END DO
      END DO
      DOFOR k = 1, n-1
            CALL LU_Pivot(a, o, s, n, k)
            IF ABS(a(o(k),k)/s(o(k))) < tol THEN
                  er = -1
                  PRINT a(o(k),k)/s(o(k))
                  EXIT DO
            END IF
            DOFOR i = k+1, n
                  factor = a(o(i),k)/a(o(k),k)
                  a(o(i),k) = factor
                  DOFOR j = k+1, n
                        a(o(i),j) = a(o(i),j) - factor*a(o(k),j)
```

```
                  END DO
            END DO
      END DO
      IF ABS(a(o(k),k)/s(o(k)) < tol THEN
            er = -1
            PRINT a(o(k),k)/s(o(k))
      END IF
END LU_Decompose

SUB LU_Pivot(a, o, s, n, k)
      p = k
      big = ABS(a(o(k),k)/s(o(k)))
      DOFOR ii = k+1, n
            dummy = ABS(a(o(ii),k)/s(o(ii)))
            IF dummy > big THEN
                  big = dummy
                  p = ii
            END IF
      END DO
      dummy = o(p) ' swap the row order
      o(p) = o(k)
      o(k) = dummy
END LU_Pivot

SUB LU_Substitute(a, n, b, x)
      DOFOR i = 2, n ' forward substitution on RHS
            sum = b(o(i))
            DOFOR j = 1, i-1
                  sum = sum -a(o(i),j)*b(o(j))
            END DO
            b(o(i)) = sum
      END DO
      x(n) = b(o(n)) / a(o(n),n)
      DOFOR i = n-1, 1, -1 ' back substitution for solution
            sum = 0
            DOFOR j = i+1, n
                  sum = sum + a(o(i),j)*x(j)
            END DO
            x(i) = (b(o(i)) - sum) / a(o(i),i)
      END DO
END LU_Substitute
```

3.2.2 Crout Decomposition

Crout Decomposition results in an upper diagonal matrix with ones on the diagonal and a lower triangular matrix. It works by simply sweeping through the matrix a single time:

$$l_{i1} = a_{i1} \quad i = 1, 2, ..., n$$

$$u_{1j} = \frac{a_{1j}}{l_{11}} \quad j = 2, 3, ..., n$$

for $j = 2, 3, ..., n\text{-}1$

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \quad i = j,\ j+1, ..., n$$

$$u_{jk} = \frac{a_{jk} - \sum_{i=1}^{j-1} l_{ji} u_{ik}}{a_{ii}} \quad k = j+1,\ j+2, ..., n$$

$$l_{nn} = a_{nn} - \sum_{k=1}^{n-1} l_{nk} u_{kn}$$

This method is particularly efficient since the coefficients of **A** are used only once as they are replaced, and the LU matrices are stored in the original matrix. Back substitution proceeds in a similar manner to that of Gauss Elimination LU decomposition, details are left to the student.

Pseudocode – Crout decomposition

```
SUB CroutDecomp(a,n)
      DOFOR j = 2, n
            a(1,j) = a(1,j) / a(1,1)
      END DO
      DOFOR j = 2, n-1
            DOFOR i = j, n
                  sum = 0
                  DOFOR k = 1, j-1
                        sum = sum + a(i,k)*a(k,j)
                  END DO
                  a(i,j) = a(i,j) - sum
            END DO
            DOFOR k = j+1, n
                  sum = 0
                  DOFOR i = 1, j-1
                        sum = sum + a(j,i)*a(i,k)
                  END DO
                  a(j,k) = (a(j,k) - sum) / a(j,j)
            END DO
      END DO
      sum = 0
      DOFOR k = 1, n-1
            sum = sum + a(n,k)*a(k,n)
      END DO
      a(n,n) = a(n,n) - sum
END CroutDecomp
```

3.2.3 Matrix Inverse

The most efficient way to determine the inverse of a matrix is to perform a LU decomposition, the successively find the columns of the inverse matrix by solving for the unit vectors

$$u_1 = \begin{Bmatrix} 1 \\ 0 \\ \dots \\ 0 \end{Bmatrix}; u_2 = \begin{Bmatrix} 0 \\ 1 \\ \dots \\ 0 \end{Bmatrix}; \dots u_n = \begin{Bmatrix} 0 \\ 0 \\ \dots \\ 1 \end{Bmatrix}$$

The matrix inverse is often used in stimulus-response studies.

3.3 Error Analysis and System Condition

A way to check an approximate solution is to substitute it into the original equations to see if the original right-hand-side results. However, this can be misleading if the system is ill-conditioned, that is, nearly singular with very large elements for $\mathbf{A}^{-1}$. A small residual $\mathbf{R} = \mathbf{B} - \mathbf{A}\bar{x}$ does not guarantee an accurate solution. Only if the largest value of $\mathbf{A}^{-1}$ is on the order of unity can the system be considered well-conditioned. Conversely, if $\mathbf{A}^{-1}$ contains elements much larger than unity we must conclude the system is ill-conditioned.

The *uniform matrix norm*, or *row-sum norm*, is

$$\|A\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$$

and can be calculated as the largest sum of the absolute values of the elements for each row. The condition number of **A** is

$$\text{cond A} = \|A\| \times \|A^{-1}\|$$

The condition number is always greater than or equal to one. It can be shown that

$$\frac{\|\Delta x\|}{\|x\|} \le \text{cond A} \frac{\|\Delta A\|}{\|A\|}$$

that is, the relative error of the solution is as large as the relative error of the norm of **A** multiplied by the condition number. If the coefficients of **A** are known to *t* digits, and the condition number of $\mathbf{A} = 10^c$, then the solution can only be valid to *t-c* digits.

Iterative refinement – In some cases, one can reduce the round-off error by substituting the approximate solution into the original set of equations and solving for correction factors.

3.4 Special Matrices

Some problem formulations result in *banded matrices*, which have elements equal to zero except for a band centered on the main diagonal. These typically occur in the solution of differential equations. Gauss Elimination and conventional LU decomposition methods can be used but are inefficient due to the many zeros. Many algorithms have been developed to solve banded systems.

3.4.1 Thomas Algorithm

Tridiagonal systems, those with a band width of 3, are most often solved with the *Thomas algorithm*. To make the algorithm particularly efficient, write the tridiagonal system as

$$\begin{bmatrix} f_1 & g_1 & & \\ e_2 & f_2 & g_2 & \\ & \cdots & & \\ & & e_n & f_n \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ \cdots \\ r_n \end{Bmatrix}$$

With this reformulation, the pseudocode below is the complete Thomas algoritm

Pseudocode – Thomas Algorithm

Decomposition

```
DOFOR k = 2, n
      e(k) = e(k) / f(k-1)
      f(k) = f(k) – e(k)*g(k-1)
END DO
```

Forward substitution

```
DOFOR k=2, n
      r(k)=r(k) - e(k)*r(k-1)
END DO
```

Back substitution

```
x(n) = r(n)/f(n)
DOFOR k = n-1, 1, -1
      x(k) = (r(k) - g(k)*x(k+1)) / f(k)
END DO
```

3.4.2 Cholesky Decomposition

A great number of engineering formulations results in symmetric, positive definite matrices. For these types of systems, the *Cholesky Decomposition* method is very efficient in that only half of the operations, and storage capacity, are necessary since $A = \mathbf{L}\mathbf{L}^T$, that is, $\mathbf{U} = \mathbf{L}^T$:

$$l_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} l_{ij} l_{kj}}{l_{ii}} \quad i = 1, 2, ..., k\text{-}1$$

$$l_{kk} = \sqrt{a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2}$$

The square root will not be a problem for a positive definite system (recall that for a positive definite system, $\mathbf{X}^T\mathbf{AX} > 0$ for all $\mathbf{X} \neq 0$).

Pseudocode – Cholesky Decomposition

```
DOFOR k = 1, n
      DOFOR i = 1, k-1
            sum = 0
            DOFOR j = 1, i-1
                  sum = sum + a(ij)*a(kj)
            END DO
            a(ki)=(a(ki) – sum) / a(ii)
      END DO
      sum = 0
      DOFOR j = 1, k-1
            sum = sum + a(kj)*a(kj)
      END DO
      a(kk) = SQRT(a(kk) – sum)
END DO
```

3.5 Iterative Techniques

Elimination techniques are the most popular way to solve systems of linear equations, but many iterative methods also exist. Iterative methods are best applied to matrices that are large and sparse, where elimination methods waste time by storing and manipulating zeros.

3.5.1 Gauss-Seidel

Given a system **Ax** = **B**, solve each of the equations for the corresponding variable

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3 - ...}{a_{11}}$$

$$x_2 = \frac{b_2 - a_{11}x_1 - a_{13}x_3 - ...}{a_{22}}$$

Given an initial guess for the values of **x**, and sweep through the system of equations and calculate new values for the x_i, always using the last calculated value of x_i. The technique is very similar to finding roots with fixed-point iteration. *Jacobi's Iteration* is a variation that calculates a complete set of x_i before using the new values; it is not as popular as Gauss-Seidel but sometimes results in better performance.

In order for Gauss-Seidel to converge, the diagonal of each row must be greater than the sum of the sum of the rest of the elements of that row, that is,

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|$$

This condition is sufficient but not necessary for convergence, that is, it will converge if it is satisfied but the solution may converge even if it is not satisfied.

In order to enhance convergence, *relaxation* is often employed. When a new x_i is calculated, modify it by a weighted average of the old and new values

$$x_i^{new} = \lambda x_i^{new} + (1-\lambda) x_i^{old}$$

where $0 \leq \lambda \leq 2$. If $0 \leq \lambda < 1$ the system is under-relaxed, which damps out oscillations in the new values. If $1 < \lambda \leq 2$ the system is over-relaxed, which speeds up convergence. Determining the value of λ for a particular problem is a matter of trial and error.

Pseudocode – Gauss-Seidel

```
SUBROUTINE GaussSeidel(a, b, n, x, imax, es, lambda)
      DOFOR i = 1, n ' normalize equations wrt diagonal element
            dummy = a(i,i)
            DOFOR j = 1, n
                  a(i,j) = a(i,j)/dummy
            END DO
            b(i) = b(i)/dummy
      END DO
      DOFOR i = 1, n
            sum = b(i)
            DOFOR j = 1, n
                  IF i<>j THEN sum = sum - a(i,j)*x(j)
            END DO
            x(i) = sum
      END DO
      iter = 1
      DO
            sentinel = 1
            DOFOR i = 1, n
                  old = x(i)
                  sum = b(i)
                  DOFOR i = 1, n
                        IF i<>j THEN sum = sum - a(i,j)*x(j)
                  END DO
                  x(i) = lambda*sum + (1 - lambda)*old
                  IF sentinel = 1 AND x(i)<>0 THEN
                        ea = ABS((x(i) - old)/x(i))
                        IF ea > es THEN sentinel = 0
                  END IF
```

```
            END DO
            iter = iter +1
            IF sentinel = 1 OR (iter >= imax) EXIT
      END DO
END GaussSeidel
```