IYTE EE204 Lecture 1
Linear Algebra Applications

Şevket Gümüştekin
(slides compiled in coop. with Barış Bozkurt)

Solving simultaneous equations

Systems of linear equations are generally expressed in matrix form. Define a matrix **A** of coefficients in the form

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ a_{20} & a_{21} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}$$

and define two vectors, **x** and **b**, with elements $x_0, x_1, , \ldots, x_{n-1}$ and $b_0, b_1, , \ldots, b_{n-1}$, respectively, then a system of linear equations can be written in the matrix form

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

Gauss-Jordan elimination

We present the method in two parts. First, we implement a simple (but slightly flawed) version of Gauss-Jordan elimination to show the basic steps of the method. In the next subsection we augment that method by a technique called *partial pivoting* to eliminate the flaw.

To see how these rules apply to a specific situation, consider a matrix **A** and a vector **b** as follows:

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & 5 \\ 1 & 2 & 4 \\ 4 & 7 & 3 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 10 \\ 5 \\ 15 \end{pmatrix}$$

This represents the following system of equations:

$$\begin{aligned} 2x_0 + 3x_1 + 5x_2 &= 10 \\ x_0 + 2x_1 + 4x_2 &= 5 \\ 4x_0 + 7x_1 + 3x_2 &= 15 \end{aligned}$$

Gauss-Jordan elimination proceeds by taking half the first equation and subtracting it from the second equation. The goal of these operations is to transform element a_{10} in the coefficient matrix **A** to zero. These transformations would yield the equations

$$\begin{aligned} 2x_0 + 3x_1 + 5x_2 &= 10 \\ 0x_0 + .5x_1 + 1.5x_2 &= 0 \\ 4x_0 + 7x_1 + 3x_2 &= 15 \end{aligned}$$

The next step would be to transform a_{20} to zero by taking twice the first equation and subtracting it from the third. This would reduce the system of equations to

$$\begin{aligned} 2x_0 + 3x_1 + 5x_2 &= 10 \\ 0x_0 + .5x_1 + 1.5x_2 &= 0 \\ 0x_0 + 1x_1 - 7x_2 &= -5 \end{aligned}$$

The effect of these transformations has been to reduce all off-diagonal coefficients in the first column of **A** to zero. We call the element on the diagonal used to transform the off-diagonal elements in the same column the *pivot element*.

The next step in Gauss-Jordan elimination is to use the diagonal element in the second row as a pivot to reduce all off-diagonal terms in the *second* column to zero. Taking 6 times the second row and subtracting it from the first row yields

$$2x_0 + 0x_1 - 4x_2 = 10$$
$$0x_0 + .5x_1 + 1.5x_2 = 0$$
$$0x_0 + 1x_1 - 7x_2 = -5$$

Taking 2 times the second row and subtracting it from the third row produces

$$2x_0 + 0x_1 - 4x_2 = 10$$
$$0x_0 + .5x_1 + 1.5x_2 = 0$$
$$0x_0 + 0x_1 - 10x_2 = -5$$

The last step is to use the third equation to eliminate off-diagonal terms in the final column. Taking .4 times the third equation and subtracting it from the first yields

$$2x_0 + 0x_1 + 0x_2 = 12$$

$$0x_0 + .5x_1 + 1.5x_2 = 0$$

$$0x_0 + 0x_1 - 10x_2 = -5$$

Taking .15 times the third equation and adding it to the second produces

$$2x_0 + 0x_1 + 0x_2 = 12$$

$$0x_0 + .5x_1 + 0x_2 = -.75$$

$$0x_0 + 0x_1 - 10x_2 = -5$$

The final solution is found by dividing the right-hand side by the diagonal coefficients, resulting in the solution $x_0 = 6$, $x_1 = -1.5$, and $x_2 = .5$. In vector notation, this solution is written as $\mathbf{x} = (6, -1.5, .5)$.

This algorithm is most easily implemented in C by creating a "working matrix" that contains all of the coefficients and the right-hand-side values. All transformations are done on the working matrix. This avoids changing values in the original arrays **A** and **b**. The working matrix has n rows and $n + 1$ columns. The first n columns are the matrix **A**, and the last column is **b**. In our simple three equation example, the working matrix would be

$$\begin{pmatrix} 2 & 3 & 5 & 10 \\ 1 & 2 & 4 & 5 \\ 4 & 7 & 3 & 15 \end{pmatrix}$$

The C implementation copies values from **A** and **b** into the working matrix and then does the appropriate Gauss-Jordan elimination. It then computes the solution vector **x**. The function requires that the user define a working array with sufficient space. This working array is one of the arguments to the function. The implementation is as follows:

```
/* Gauss-Jordan without partial pivoting */
#define A(I,J) (*(a + (I)*n + (J)))
#define WORK(I,J) (*(work + (I)*(n+1) + (J)))

void simple_gauss(
  double *a,      /* the coefficients in A */
  double b[],     /* the right-hand side b */
  double *work,   /* the working array */
  double x[],     /* the solution array */
  int n)          /* number of equations */

{
  double m;
  int i,j,k;

/* set up working matrix */
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++)
      WORK(i,j) = A(i,j);
    WORK(i,n) = b[i];
  }
```

```
/* loop through the rows of the working matrix */
  for(i=0; i<n; i++)
/* perform elimination */
    for(j=0; j<n; j++)
      if(j != i) {
        m = WORK(j,i)/WORK(i,i);
        for(k=i; k<=n; k++)
          WORK(j,k) -= m*WORK(i,k);
      }

/* compute solution as rhs divided by diagonal */

  for(i=0; i<n; i++)
    x[i] = WORK(i,n)/WORK(i,i);
}

#undef A
#undef WORK
```

The function prototype for **simple_gauss()** is

```
void simple_gauss(double *, double *, double *,
                  double *, int);
```

and a code fragment that uses the function to solve our sample three-equation linear system would be

```
/* code using simple_gauss() */
double coeff[3][3] = { {2,3,5}, {1,2,4}, {4,7,3}};
double rhs[3] = {10,5,15};
double place[3][4];
double answer[3];
simple_gauss(&coeff[0][0],rhs,&place[0][0],answer, 3);
printf("Solution is: %lf %lf %lf\n", answer[0],
          answer[1], answer[2]);
```

Note that the two matrices, **coeff** and **place**, are passed to **simple_gauss()** in the form **&coeff[0][0]** and **&place[0][0]** rather than as **coeff** and **mat**. This is because, as discussed in Section 4.6.2, **coeff** and **mat** are not pointers to **double**s, but are pointers to *arrays* of **double**s. The function **simple_gauss()** is written to accept pointers to **double**s, which are then used as the base addresses of the matrices. Many C compilers will accept either form for the arguments even though the use of the matrix names alone is technically incorrect.

4.8.2 Partial Pivoting

The implementation of `simple_gauss()` unfortunately has a major flaw. *In particular, the method fails if any of the pivot elements is zero.* In this case the algorithm will attempt to divide an element of the coefficient matrix by zero, producing a floating-point error.

In order to avoid division by zero, we need to select a nonzero pivot element at each step. The simplest method is based on the observation that exchanging two rows in the working matrix leaves the system of equations unchanged. Thus, before pivoting on a diagonal element, we can look for a row in the working matrix that would produce a nonzero pivot element. Once we find this row, we can exchange it with the row that would have yielded a zero pivot value. This approach is called *partial pivoting*.

At each step of Gauss-Jordan elimination, all the rows below the one currently being considered are candidates for pivot elements. The rows above the current one already have been used for pivot elements. They cannot be used again because moving them to a different row of the working matrix will place their nonzero diagonal element in a nondiagonal position.

In general, there will be more than one candidate row that yields a nonzero pivot element. The selection of which one to use turns out to be significant. This is because without an appropriate selection, it is possible to show that Gauss-Jordan elimination is numerically unstable. Roundoff errors tend to accumulate in ways that seriously compromise the accuracy of the method.

The basic steps of partial pivoting are as follows:

- **Step 1.** Set up a working matrix as in the simple version of Gauss-Jordan elimination.
- **Step 2.** For each row i (starting with row 0) in the working matrix:
  1. Check rows i through $n-1$ for the one which yields the pivot element with the largest absolute value. Call that row k.
  2. Swap row i with k.
  3. Perform Gauss-Jordan elimination using element ii of the working matrix as the pivot.
- **Step 3.** For each row i compute solution x_i as element in of the working matrix divided by element ii of the working matrix.

```
/* implementation of Gauss-Jordan elimination
   with partial pivoting */

#include <math.h>
#define TRUE 1

#define FALSE 0
#define A(I,J) (*(a + (I)*n + (J)))
#define WORK(I,J) (*(work + (I)*(n+1) + (J)))

int gauss_jordan(
  double *a,      /* the coefficients in A */
  double b[],     /* the right-hand side b */
  double *work,   /* the working array */
  double x[],     /* the solution array */
  int n,          /* number of equations */
  double tol)     /* minimum allowed pivot element */

{
  double m, max, temp;
  int i,j,k, swap;
```

```
/* set up working matrix */
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++)
      WORK(i,j) = A(i,j);
    WORK(i,n) = b[i];
  }

/* loop through the rows of the working matrix */
  for(i=0; i<n; i++) {
    max = -1.0;
    for(k=i; k<n; k++)
      if(fabs(WORK(k,i)) > max) {
        max = fabs(WORK(k,i));
        swap = k;
      }
/* check if pivot element is "large enough" */
    if(max <= tol)
      return(FALSE);
```

```
  /* swap rows */
      if(swap != i)
        for(k=i; k<=n; k++) {
  temp = WORK(i,k);
          WORK(i,k) = WORK(swap,k);
          WORK(swap,k) = temp;
        }

  /* perform elimination */
      for(j=0; j<n; j++)
        if(j != i) {
          m = WORK(j,i)/WORK(i,i);
          for(k=i; k<=n; k++)
            WORK(j,k) -= m*WORK(i,k);
        }
   }
/* compute solution as rhs divided by diagonal */

   for(i=0; i<n; i++)
     x[i] = WORK(i,n)/WORK(i,i);
   return(TRUE);
}

#undef A
#undef WORK
```

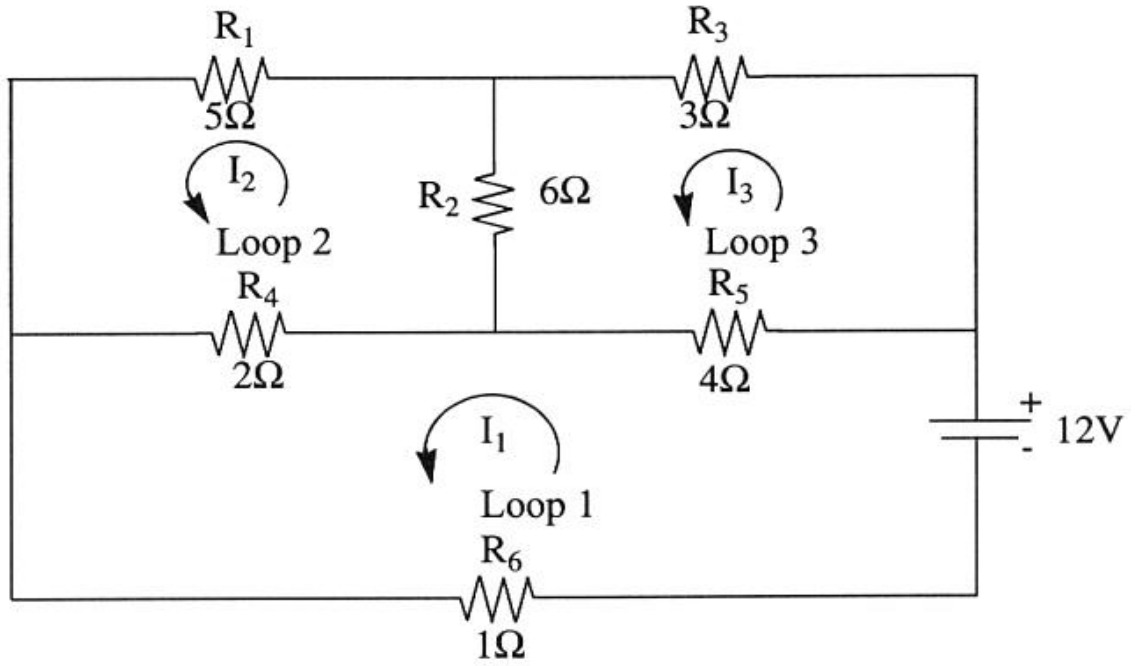
The function **gauss_jordan()** returns an **int** which is **TRUE** if it found a solution without encountering a pivot element smaller than the tolerance **tol**, and returns **FALSE** otherwise. A typical usage for the function is shown in the following **main()** program.

```
/* example program using Gauss-Jordan elimination */
#include <stdio.h>
int gauss_jordan(double *, double *, double *,
                 double *, int, double);

double coeff[3][3] = { {2,3,5}, {1,2,4}, {4,7,3}};
double rhs[3] = {10,5,15};
double place[3][4];

main()
{
  double answer[3];
  if (gauss_jordan(&coeff[0][0], rhs, &place[0][0],
                   answer, 3, .00001))
    printf("Solution is: %lf %lf %lf\n",
                   answer[0], answer[1], answer[2]);
  else
    printf("No solution found\n");
}
```

Application: Electrical circuit networks



For loop 1:

$$R_4(I_1 - I_2) + R_5(I_1 - I_3) + R_6 I_1 = 12$$

For loop 2:

$$R_1 I_2 - R_2(I_2 - I_3) + R_4(I_2 - I_1) = 0$$

For loop 3:

$$R_2(I_3 - I_2) + R_3 I_3 - R_5(I_3 - I_1) = 0$$

$$7I_1 - 2I_2 - 4I_3 = 12$$

$$-2I_1 + 13I_2 - 6I_3 = 0$$

$$-4I_1 - 6I_2 + 13I_3 = 0$$

```
      SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS
       USING GAUSS ELIMINATION


  This program uses Gauss Elimination to solve the
  system [A]{X} = {B}, where [A] is the matrix of known
  coefficients, {B} is the vector of known constants,
   and {X} is the column matrix of the unknowns.
Number of equations: 3

Enter elements of matrix [A]
A(1,1) = 7
A(1,2) = -2
A(1,3) = -4
A(2,1) = -2
A(2,2) = 13
A(2,3) = -6
A(3,1) = -4
A(3,2) = -6
A(3,3) = 13

Enter elements of [B] vector
B(1) = 12
B(2) = 0
B(3) = 0

      SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS

       The solution is
       x(1) = 2.775652
       x(2) = 1.043478
       x(3) = 1.335652
       Determinant = 575.000000
```

Iterative methods for solving linear equations

Iterative methods are also available where initial guess is made for solution and then improved in steps. Unlike direct methods, iterative methods may not always yield a solution.

$$
\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_1 & & \ldots & + & a_{2n}x_n & = & b_2 \\
. & & . & & . & & . & & . \\
. & & . & & . & & . & & . \\
. & & . & & . & & . & & . \\
a_{n1}x_1 & + & a_{n2}x_2 & & & + & a_{nn}x_n & = & b_n
\end{array}
$$

is rewritten as

$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}x_n)/a_{11}$$

$$x_2 = (b_2 - a_{21}x_1 - a_{23}x_3 - \ldots - a_{2n}x_n)/a_{22}$$

$$\vdots$$

$$x_n = (b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{nn-1}x_{n-1})/a_{nn}$$

An iterative procedure is guaranteed to converge for any initial guess for the solution if the coefficient matrix **A** is diagonally dominant. A diagonally dominant matrix is one in which the absolute value of the diagonal element a_{ii} in each row i is larger than the sum of the absolute values of all other elements in that row. That is, for each row $i = 1, 2, ..., n$

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

An iterative method may still converge even if the coefficient matrix is not diagonally dominant, particularly if the largest elements of the matrix are located on the diagonal. Thus, the requirement of diagonal dominance is a sufficient condition but not a necessary condition and it is possible to find a solution to systems that possess a strong diagonal.

THE JACOBI METHOD

One of the simplest iterative methods is the Jacobi method. As with the other iterative methods, the Jacobi method begins by making an initial guess of the values of $x_1^{(1)}$ $x_2^{(1)}$,..., $a_n^{(1)}$. A common approach is to use the following first approximation:

$$x_1^{(1)} = b_1/a_{11}$$

$$x_2^{(1)} = b_2/a_{22}$$

.

.

.

$$x_n^{(1)} = b_n/a_{nn}$$

The superscript in the above equations represents the number of iterations performed in approximating the solution.

The second approximation is obtained by substituting these first approximations in Equation 19.14. Thus,

$$x_1^{(2)} = (b_1 - a_{12}x_2^{(1)} - a_{13}x_3^{(1)} - \ldots - a_{1n}x_n^{(1)})/a_{11}$$

$$x_2^{(2)} = (b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(1)} - \ldots - a_{2n}x_n^{(1)})/a_{22}$$

$$\vdots$$

$$x_n^{(2)} = (b_n - a_{n1}x_1^{(1)} - a_{n2}x_2^{(1)} - \ldots - a_{nn-1}x_{n-1}^{(1)})/a_{nn} \qquad \textbf{(19.17)}$$

The equations used to solve for the x_i's for the kth iteration can be written as:

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \ldots - a_{1n}x_n^{(k)})/a_{11}$$

$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \ldots - a_{2n}x_n^{(k)})/a_{22}$$

$$\vdots$$

$$x_n^{(k+1)} = (b_n - a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \ldots - a_{nn-1}x_{n-1}^{(k)})/a_{nn} \qquad \textbf{(19.18)}$$

In general, for a n by n system of equations, these equations can be written as

$$x_i^{(k+1)} = b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}x_i^{(k)} \qquad \textbf{(19.19)}$$

The process is continued until the new values $\{x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_n^{(k+1)}\}$ are sufficiently close to the previous values $\{x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}\}$. The test for convergence can be written as

$$\left|x_i^{(k+1))} - x_i^{(k)}\right| > \left|x_i^{(k+1)} \cdot \varepsilon\right| \quad , \qquad i = 1, 2, \ldots, n \tag{19.20}$$

where ε is the desired tolerance.

Example 19.5 Jacobi Method

PROBLEM STATEMENT: Solve the following set of linear simultaneous equations using the Jacobi method

$$\begin{aligned} 10x_1 + 2x_2 + 3x_3 &= 11 \\ x_1 + 5x_2 + 2x_3 &= 20 \\ 3x_1 + 2x_2 + 6x_3 &= -12 \end{aligned}$$

SOLUTION: We begin by first checking the diagonal coefficients to determine if the coefficient matrix is diagonally dominant.

$|10| > |2| + |3| \rightarrow 10 > 5$ O.K.

$|5| > |1| + |2| \rightarrow 5 > 3$ O.K.

$|6| > |3| + |2| \rightarrow 6 > 5$ O.K.

The coefficient matrix is diagonally dominant so the Jacobi iterative procedure will converge.

We compute initial values for $x_1^{(1)}$, $x_2^{(1)}$, and $x_3^{(1)}$ from

$$x_1^{(1)} = \frac{b_2}{a_{11}} = \frac{10}{10} = 1.0$$

$$x_2^{(1)} = \frac{b_2}{a_{22}} = \frac{20}{5} = 4.0$$

$$x_3^{(1)} = \frac{b_3}{a_{33}} = -\frac{12}{6} = -2.0$$

The second approximation to the solution is

$$x_1^{(2)} = \frac{(b_1 - a_{12}x_2^{(1)} - a_{13}x_3^{(1)})}{a_{11}} = \frac{[10 - 2(4) - 3(-2.0)]}{10} = 0.8$$

$$x_2^{(2)} = \frac{(b_2 - a_{21}x_1^{(1)} - a_{23}x_3^{(1)})}{a_{22}} = \frac{[20 - 1(1.0) - 2(-2.0)]}{5} = 4.6$$

$$x_3^{(2)} = \frac{b_3 - a_{31}x_1^{(1)} - a_{32}x_2^{(1)}}{a_{33}} = \frac{-12 - 3(1.0) - 2(4.0)}{6} = -3.833$$

Substituting the above values, we obtain the third approximation from

$$x_1^{(3)} = \frac{b_1 - a_{12}x_2^{(2)} - a_{13}x_3^{(2)}}{a_{11}} = \frac{10 - 2(4.6) - 3(-3.833)}{10} = 1.230$$

$$x_2^{(3)} = \frac{b_2 - a_{21}x_1^{(2)} - a_{23}x_3^{(2)}}{a_{22}} = \frac{20 - 1(0.8) - 2(-3.833)}{5} = 5.373$$

$$x_3^{(3)} = \frac{b_3 - a_{31}x_1^{(2)} - a_{32}x_2^{(2)}}{a_{33}} = \frac{-12 - 3(0.8) - 2(4.6)}{6} = -3.933$$

Table 19.4 shows the results for the first seven iterations. It is left for the reader to verify that the solution is $x_1 = 1.231$, $x_2 = 5.538$, and $x_3 = -4.462$.

Table 19.4: Results of first seven iteration for the Jacobi method

Variable	Iteration Number						
	1	2	3	4	5	6	7
x_1	1.000	0.800	1.230	1.105	1.256	1.190	1.247
x_2	4.000	4.600	5.373	5.373	5.480	5.480	5.552
x_3	-2.000	-3.833	-3.933	-4.406	-4.328	-4.475	4.422

C program for the Jacobi method

```
#include <stdio.h>
#include <math.h>

#define MAXSIZE 20                          /* maximum size of matrix */
#define FALSE    0
#define TRUE     1
/* function prototype */
int jacobi(double a[][MAXSIZE],double b[],int n,int max_iter,
           double tol,double x[]);
```

```
void main(void)
{
   double a[MAXSIZE][MAXSIZE],b[MAXSIZE],x[MAXSIZE];
   double tol;
   int i,j,n,return_val;
   int max_iter;

   printf("\n \t SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS");
   printf("\n \t USING THE JACOBI METHOD\n\n ");
   printf("\n    This program uses the JAcobi Method to solve the");
   printf("\n    system [A]{X} = {B}, where [A] is the matrix of");
   printf("\n    known coefficients, {B} is the vector of known
          constants,");
   printf("\n    and {X} is the column matrix of the unknowns.");

   /* get number of equations */
   n = 0;
   while (n <= 0 || n > MAXSIZE)
      {
      printf("\n Number of equations: ");
      scanf("%d", &n);
      }
   /* Read matrix [A] */
   printf("\n Enter elements of matrix [A] \n");
   for (i=0; i < n; ++i)
       for (j=0; j < n; ++j)
           {
           printf(" A(%d,%d) = ", i+1,j+1);
           scanf("%lf", &a[i][j]);
           }
   /*  Read {B} vector    */
   printf("\n Enter elements of [B] vector \n");
   for (i = 0; i < n; ++i)
       {
       printf(" B(%d) = ",i+1);
       scanf("%lf", &b[i]);
       }
```

```
    /* read maximum number of iterations */
    printf("\n Maximum number of iterations: ");
    scanf("%d", &max_iter);

    /* read tolerance */
    printf("\n Enter desired tolerance ");
    scanf("%d", &tol);

    /* call Jacobi function */
    return_val= jacobi(a,b,n,max_iter,tol,x);

    /* print results     */
    if (return_val == 0)
       {
       printf("\n \t SOLUTION OF SIMULTANEOUS LINEAR EQUATIONS\n");
       printf("\n \t The solution is");
       }
    else
       {
       printf("\n\n\t Did not converge after %d iterations",max_iter);
       printf("\n Current values of x{} are ");
       }
    for (i=0; i < n; ++i)
       printf("\n \t x(%d) = %lf",i+1,x[i]);
}
```

```
/*-------------------------------------------------------------*/
/*  Function: jacobi()                                         */
/*                                                             */
/*   This function solves the system of equations              */
/*   [A]{x} = {B} using the Jacobi iterative method.           */
/*                                                             */
/*   Input Parameters:                                         */
/*       n              - number of equations                  */
/*       a[n][n]        - coefficient matrix                   */
/*       b[n]           - right-hand side vector               */
/*       max_iter       - maximum number of iterations         */
/*                                                             */
/*   Output Parameters:                                        */
/*       x[n]           - solution vector                      */
/*                                                             */
/*   Returns                                                   */
/*      0 if successful                                        */
/*      1 if did not converge after max_iter iterations        */
/*                                                             */
/*-------------------------------------------------------------*/
int jacobi(double a[][MAXSIZE],double b[],int n,int max_iter,
           double tol,double x[])
{
  int num_iter=0;
  int tol_exceeded = TRUE;
  int i,j;
  double x_old[MAXSIZE];        /* previous solution */
  double sum;
```

```
  for (i=0; i < n; ++i)
     x[i] = b[i]/a[i][i];

  while (tol_exceeded && num_iter < max_iter)
     {
     /* save old values of x{} */
     for (i=0; i < n; ++i)
        x_old[i] = x[i];

     /* compute new values of x{} */

     for (i = 0; i < n; ++i)
        {
        sum = b[i];
        for (j = 0; j <n; ++j)
           if (i != j )
             sum -= a[i][j] * x_old[j];

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

    /* check if solution is within desired tolerance */
      tol_exceeded = FALSE;
    for (i = 0; i < n ; ++i)
       if ( fabs(x[i] - x_old[i]) > fabs(x_old[i] * tol) )
         tol_exceeded = TRUE;

    ++num_iter;
    }

  return(tol_exceeded);
}
```

Matrix Inversion

Note that, using the inverse of a square matrix A, one can get:

$$AA^{-1} = I$$

Gauss-Jordan elimination can be used for finding the inverse of a matrix by restructuring the above equation:

$$\begin{aligned} Ax^0 &= I^0 \\ Ax^1 &= I^1 \\ Ax^2 &= I^2 \\ &\vdots \\ Ax^{n-1} &= I^{n-1} \end{aligned}$$

Where x^i and I^i are the respective columns of inverse matrix A^{-1} and identity matrix I .