Matrix Operations with SCILAB

By

Gilberto E. Urroz, Ph.D., P.E.

Distributed by

*i*nfoClearinghouse.com

A "zip" file containing all of the programs in this document (and other SCILAB documents at InfoClearinghouse.com) can be downloaded at the following site:

http://www.engineering.usu.edu/cee/faculty/gurro/Software_Calculators/Scilab_Docs/ScilabBookFunctions.zip

The author's SCILAB web page can be accessed at:

http://www.engineering.usu.edu/cee/faculty/gurro/Scilab.html

Please report any errors in this document to: gurro@cc.usu.edu

Gaussian and Gauss-Jordan elimination

Gaussian elimination is a procedure by which the square matrix of coefficients belonging to a system of *n* linear equations in *n* unknowns is reduced to an upper-triangular matrix (*echelon form*) through a series of row operations. This procedure is known as *forward elimination*. The reduction of the coefficient matrix to an upper-triangular form allows for the solution of all *n* unknowns, utilizing only one equation at a time, in a procedure known as *backward substitution*.

Gaussian elimination using a system of equations

To illustrate the Gaussian elimination procedure we will use the following system of 3 equations in 3 unknowns:

$$\begin{aligned} 2X +4Y+6Z &= 14, \\ 3X -2Y+ Z &= -3, \\ 4X +2Y -Z &= -4. \end{aligned}$$

Forward elimination

First, we replace the second equation by (equation 2 – (3/2)* equation 1), and the third by (equation 1 – (4/2)*equation 1), to get

$$\begin{aligned} 2X +4Y+6Z &= 14, \\ -8Y-8Z &= -24, \\ -6Y-13Z &= -32. \end{aligned}$$

Next, replace the third equation with (equation 3 - (6/8)*equation 2), to get

$$\begin{aligned} 2X +4Y+6Z &= 14, \\ -8Y-8Z &= -24, \\ -7Z &= -14. \end{aligned}$$

Backward substitution

The system of equations is now in an upper-triangular form, allowing us to solve for Z first, as

$$Z = -14/-7 = 2.$$

We then replace this value into equation 2 to solve for Y, as

$$Y = (-24 +8Z)/(-8) = (-24 +8\cdot 2)/(-8) = 1.$$

Finally, we replace the values of Y and Z into equation 1 to solve for X as

$$X = (14-4Y-6Z)/2 = (14-4(1)-6(2))/2 = -1.$$

The solution is, therefore, X = -1, Y = 1, Z = 2.

Gaussian elimination using matrices

The system of equations used in the example above, i.e.,

$$\begin{aligned} 2X + 4Y + 6Z &= 14, \\ 3X - 2Y + Z &= -3, \\ 4X + 2Y - Z &= -4. \end{aligned}$$

Can be written as a matrix equation $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$, if we use:

$$\mathbf{A} = \begin{pmatrix} 2 & 4 & 6 \\ 3 & -2 & 1 \\ 4 & 2 & -1 \end{pmatrix}, \quad \mathbf{x} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 14 \\ -3 \\ -4 \end{bmatrix}.$$

To obtain a solution to the system matrix equation using Gaussian elimination, we first create what is known as the *augmented matrix* corresponding to **A**, i.e.,

$$\mathbf{A}_{aug} = \left(\begin{array}{ccc|c} 2 & 4 & 6 & 14 \\ 3 & -2 & 1 & -3 \\ 4 & 2 & -1 & -4 \end{array}\right).$$

The matrix $\mathbf{A}_{aug}$ is nothing more than the original matrix A with a new row, corresponding to the elements of the vector **b**, added (i.e., augmented) to the right of the rightmost column of **A**.

Once the augmented matrix is put together, we can proceed to perform row operations on it that will reduce the original A matrix into an upper-triangular matrix similar to what we did with the system of equations shown earlier. If you perform the forward elimination by hand, you would write the following:

$$\mathbf{A}_{aug} = \left(\begin{array}{ccc|c} 2 & 4 & 6 & 14 \\ 3 & -2 & 1 & -3 \\ 4 & 2 & -1 & -4 \end{array}\right) \cong \left(\begin{array}{ccc|c} 2 & 4 & 6 & 14 \\ 0 & -8 & -8 & -24 \\ 0 & -6 & -13 & -32 \end{array}\right) \cong \left(\begin{array}{ccc|c} 2 & 4 & 6 & 14 \\ 0 & -8 & -8 & -24 \\ 0 & 0 & -7 & -14 \end{array}\right)$$

The symbol ≅ (" is congruent to") indicates that what follows is equivalent to the previous matrix with some row (or column) operations involved.

After the augmented matrix is reduced as shown above, we can proceed to perform the backward substitution by converting the augmented matrix into a system of equations and solving for x_3, x_2, and x_1 in that order, as performed earlier.

Gaussian elimination algorithm

In this section we present the algorithm for performing Gaussian elimination on an augmented matrix. The augmented matrix results from the system of linear equations represented by the matricial equation **A**·**x** = **b**. The matrix **A** and vectors **x** and **b** are given by

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

The augmented matrix is:

$$A_{aug} = \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1m} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2m} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} & b_n \end{array}\right].$$

Thus, the augmented matrix can be referred to as $(\mathbf{A}_{aug})_{nx(n+1)} = [a_{ij}]$, with $a_{i,n+1} = b_i$, for $i = 1,2, \ldots, n$.

Augmenting a matrix in SCILAB

In SCILAB augmenting a matrix **A** by a column vector **b** is straightforward. For example, for the linear system introduced earlier, namely,

$$\mathbf{A} = \begin{pmatrix} 2 & 4 & 6 \\ 3 & -2 & 1 \\ 4 & 2 & -1 \end{pmatrix}, \quad \mathbf{b} = \begin{bmatrix} 14 \\ -3 \\ -4 \end{bmatrix},$$

an augmented matrix, $\mathbf{A}_{aug}$, is obtained as follows:

```
-->A = [2,4,6;3,-2,1;4,2,-1], b = [1;-3;-4]
 A  =

!   2.    4.    6. !
!   3.  - 2.    1. !
!   4.    2.  - 1. !
```



```
 b  =

!   1. !
! - 3. !
! - 4. !

-->A_aug = [A b]
 A_aug  =

!   2.    4.    6.    1. !
!   3.  - 2.    1.  - 3. !
!   4.    2.  - 1.  - 4. !
```

Algorithm for first step in forward elimination

After the augmented matrix has been created the first elimination pass will fill with zeros those values in the first column below a_{11}, i.e.,

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1m} & a_{1,n+1} \\ 0 & a^*_{22} & \cdots & a^*_{2m} & a^*_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a^*_{n2} & \cdots & a^*_{nm} & a^*_{n,n+1} \end{array}\right].$$

The procedure is indicated by the expressions:

$a^*_{21} = a^*_{31} = \ldots = a^*_{n1} = 0$; j = 1 (first column)

$a^*_{2,j} = a_{2j} - a_{1j} \cdot a_{21}/a_{11}$; j = 2,3, . . . , n+1
$a^*_{3j} = a_{3,j} - a_{1j} \cdot a_{31}/a_{11}$; j = 2,3, . . . , n+1
. . .
. . .
. . .
$a^*_{n,j} = a_{n,j} - a_{1j} \cdot a_{n1}/a_{11}$; j = 2,3, . . . , n+1

or, more concisely, as

$a^*_{ij} = a_{ij} - a_{1j} \cdot a_{i1}/a_{11}$; for i = 2,3, …n; and, j = 2,3,. . .,n+1
$a^*_{ij} = 0$; for i = 2,3, …, n; and, j = 1

SCILAB example for first step in forward elimination

To illustrate this first step using SCILAB, we first copy the augmented matrix we created into an array *a* by using:

```
-->a = A_aug
 a  =
```



```
!   2.    4.    6.    1. !
!   3.  - 2.    1.  - 3. !
!   4.    2.  - 1.  - 4. !
```

Next, we define n as 3 and implement the first step in the forward elimination through the use of *for..end* constructs:

```
--> n = 3;
-->for i=2:n, for j=2:n+1, a(i,j)=a(i,j)-a(1,j)*a(i,1)/a(1,1); end; a(i,1) = 0;
end;
```

To see the modified augmented matrix we use:

```
-->a
 a  =

!   2.    4.    6.     14. !
!   0.  - 8.  - 8.   - 24. !
!   0.  - 6.  - 13.  - 32. !
```

Algorithm for second step in forward elimination

After the second elimination pass, the generic augmented matrix is:

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & a_{1,n+1} \\ 0 & a^*_{22} & a^*_{23} & \cdots & a^*_{2n} & a^*_{2,n+1} \\ 0 & 0 & a^{**}_{33} & \cdots & a^{**}_{3n} & a^{**}_{3,n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & a^{**}_{n3} & \cdots & a^{**}_{nn} & a^{**}_{n,n+1} \end{array}\right]$$

with rows i = 3, 4, …n, recalculated as:

$a^{**}_{ij} = a^*_{ij} - a^*_{2j} \cdot a^*_{i2}/a^*_{22}$; for i = 3,4, …n; and, j = 3,4,. . .,n+1
$a^{**}_{ij} = 0$; for i = 3,4,…,n; and, j = 2

SCILAB example for second step in forward elimination

For the SCILAB example under consideration, this second step will be the final step in the forward elimination. This step is implemented as follows:

```
--> for i=3:n, for j=3:n+1, a(i,j)=a(i,j)-a(2,j)*a(i,2)/a(2,2); end; a(i,2) =
0; end;
```


The result is:

```
a  =

!   2.    4.    6.    14. !
!   0.  - 8.  - 8.  - 24. !
!   0.    0.  - 7.  - 14. !
```

Algorithm for third and subsequent steps in the forward elimination

For a general *nx(n+1)* augmented matrix, we can predict that in the next (third) elimination pass the new coefficients in the augmented matrix can be written as:

$$a^{***}_{ij} = a^{**}_{ij} - a^{**}_{3j} \cdot a^{**}_{i3}/a^{**}_{33};\ \text{for i = 4,5, ...n; and, j = 4,5,. . .,n+1}$$
$$a^{***}_{ij} = 0;\ \text{for i = 4,5,...,n; and, j = 3}$$

Notice that the number of asterisks in the new coefficients calculated for each elimination pass corresponds to the number of the elimination pass, i.e., we calculated a^{*}_{ij} in the first pass, a^{**}_{ij} in the second pass, and so on. We may venture to summarize all the equations we have written for the new coefficients as:

$$a^{(k)}_{ij} = a^{(k-1)}_{ij} - a^{(k-1)}_{kj} \cdot a^{(k-1)}_{ik}/a^{(k-1)}_{kk};\ \text{for i = k+1,k+2, ...n; and, j = k+1,k+2,. . .,n+1,}$$
$$a^{(k)}_{ij} = 0;\ \text{for i = k+1,k+2,...,n; and, j = 1,2,...,k}$$

$$\text{where k = 1,2,. . .,n-1.}$$

Notice that k represents the order of the elimination pass, i represents the row index, and j represents the column index.

After the *forward elimination* passes have been completed, the modified augmented matrix should look like this:

$$\left[\begin{array}{ccccccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1,n-1} & a_{1n} & a_{1,n+1} \\ 0 & a^{(1)}_{22} & a^{(1)}_{23} & \cdots & a^{(1)}_{2,n-1} & a^{(1)}_{2n} & a^{(1)}_{2,n+1} \\ 0 & 0 & a^{(2)}_{33} & \cdots & a^{(2)}_{3,n-1} & a^{(2)}_{3n} & a^{(2)}_{3,n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a^{(n-2)}_{n-1,n-1} & a^{(n-2)}_{n-1,n} & a^{(n-2)}_{n-1,n+1} \\ 0 & 0 & 0 & \cdots & 0 & a^{(n-1)}_{n,n} & a^{(n-1)}_{n,n+1} \end{array}\right].$$

If we use the array a_{ij} to store the modified coefficients, we can simply write:

$$a_{ij} = a_{ij} - a_{kj} \cdot a_{ik}/a_{kk};\ \text{i = k+1,k+2,...,n; j = k+1,k+2,...,n+1.}$$
$$a_{ij} = 0;\ \text{for i = k+1,k+2,...,n; and, j = k}$$

for k = 1,2,...,n-1.

Note: By using the array a_{ij} to store the newly calculated coefficients you loose the information stored in the original array a_{ij}. Therefore, when implementing the algorithm in SCILAB, if you need to keep the original array available for any reason, you may want to copy it into a different array as we did in the SCILAB example presented earlier.

SCILAB example implementing the full forward elimination algorithm

For the 3x4 augmented matrix presented earlier, the following SCILAB commands implement the algorithm for forward elimination:

```
-->a = A_aug
 a  =

!   2.    4.    6.    1. !
!   3.  - 2.    1.  - 3. !
!   4.    2.  - 1.  - 4. !

-->n=3;

-->for k=1:n-1, for i=k+1:n, for j=k+1:n+1, a(i,j)=a(i,j)-a(k,j)*a(i,k)/a(k,k);
end; for j = 1:k, a(i,j) = 0; end; end; end;


-->a
 a  =

!   2.    4.    6.    14. !
!   0.  - 8.  - 8.  - 24. !
!   0.    0.  - 7.  - 14. !
```

Algorithm for backward substitution

The next step in the algorithm is to calculate the solution $x_1, x_2, \ldots, x_n$, by *back substitution*, starting with

$$x_n = a_{n,n+1}/a_{nn}.$$

and continuing with

$$x_{n-1} = (a_{n-1,n+1}-a_{n-1,n}x_n)/a_{n-1,n-1}$$

$$x_{n-2} = (a_{n-2,n+1} - a_{n-2,n-1}x_{n-1} - a_{n-2,n}x_n)/a_{n-2,n-2}$$

.

.

$$x_1 = \frac{a_{1,n+1} - \sum_{k=2}^{n} a_{1k} x_k}{a_{11}}.$$

In summary, the solution is obtained by first calculating:

$$x_n = a_{n,n+1}/a_{nn}.$$

and, then, calculating

$$x_i = \frac{a_{i,n+1} - \sum_{k=i+1}^{n} a_{ik} x_k}{a_{ii}},$$

for i = n-1, n-2, ..., 2,1 (i.e., counting backwards from (n-1) to 1).

SCILAB example for calculating the unknown x's

Using the current value of the matrix *a* that resulted from the forward elimination, we can calculate the unknowns in the linear system as follows. First, the last unknown is:

```
-->x(n) = a(n,n+1)/a(n,n);
```

Then, we calculate the remaining unknowns using:

```
-->for i = n-1:-1:1, sumk=0; for k=i+1:n, sumk=sumk+a(i,k)*x(k); end;
x(i)=(a(i,n+1)-sumk)/a(i,i); end;
```

The solution is:

```
-->x
 x  =

! - 1. !
!   1. !
!   2. !
```

SCILAB function for Gaussian elimination

The following SCILAB function implements the solution of the system of linear equations, **A**·**x**=**b**. The function's arguments are a nxn matrix A and a nx1 vector b. The function returns the solution x.

```
function [x] = gausselim(A,b)

//This function obtains the solution to the system of
//linear equations A*x = b, given the matrix of coefficients A
//and the right-hand side vector, b

[nA,mA] = size(A)
[nb,mb] = size(b)

if nA<>mA then
                        error('gausselim - Matrix A must be square');
                        abort;
elseif mA<>nb then
        error('gausselim - incompatible dimensions between A and b');
        abort;
end;

a = [A b];  //Matrix augmentation

//Forward elimination

n = nA;
for k=1:n-1
    for i=k+1:n
                        for j=k+1:n+1
                          a(i,j)=a(i,j)-a(k,j)*a(i,k)/a(k,k);
                        end;
    end;
end;

//Backward substitution

x(n) = a(n,n+1)/a(n,n);

for i = n-1:-1:1
                        sumk=0
                        for k=i+1:n
                          sumk=sumk+a(i,k)*x(k);
                        end;
                        x(i)=(a(i,n+1)-sumk)/a(i,i);
end;

//End function
```

Note: In this function we did not include the statements that produce the zero values in the lower triangular part of the augmented matrix. These terms are not involved in the solution at all, and were used earlier only to illustrate the effects of the Gaussian elimination.

Application of the function *gausselim* to the problem under consideration produces:

```
->A = [2,4,6;3,-2,1;4,2,-1]; b = [14;-3;-4];

-->getf('gausselim')  //Loading the function
```



```
-->gausselim(A,b)    //Function call without assignment
 ans  =

! - 1. !
!   1. !
!   2. !

-->x = gausselim(A,b)  //Function call with assignment
 x  =

! - 1. !
!   1. !
!   2. !
```

This simple implementation of Gaussian elimination is commonly referred to as "naïve Gaussian elimination," because it does not account for the possibility of having zero terms in the diagonal. Such terms will produce a division by zero in the Gaussian elimination algorithm effectively terminating the process in SCILAB. A technique known as "partial pivoting" can be implemented to account for the case of zero diagonal terms. Details on pivoting are presented next.

Pivoting

If you look carefully at the row operations in the examples shown above, you will notice that many of those operations divide a row by its corresponding element in the main diagonal. This element is called a pivot element, or simply, a *pivot*. In many situations it is possible that the pivot element become zero, in which case a division by zero occurs. Also, to improve the numerical solution of a system of equations using Gaussian or Gauss-Jordan elimination, it is recommended that the pivot be the element with the largest absolute value in a given column. This operation is called *partial pivoting*. To follow this recommendation is it often necessary to exchange rows in the augmented matrix while performing a Gaussian or Gauss-Jordan elimination.

While performing pivoting in a matrix elimination procedure, you can improve the numerical solution even more by selecting as the pivot the element with the largest absolute value in the column and row of interest. This operation may require exchanging not only rows, but also columns, in some pivoting operations. When row and column exchanges are allowed in pivoting, the procedure is known as *full pivoting*.

When exchanging rows and columns in partial or full pivoting, it is necessary to keep track of the exchanges because the order of the unknowns in the solution is altered by those exchanges. One way to keep track of column exchanges in partial or full pivoting mode, is to create a *permutation matrix* $\mathbf{P} = \mathbf{I}_{n\times n}$, at the beginning of the procedure. Any row or column exchange required in the augmented matrix $\mathbf{A}_{aug}$ is also registered as a row or column exchange, respectively, in the permutation matrix. When the solution is achieved, then, we multiply the permutation matrix by the unknown vector $\mathbf{x}$ to obtain the order of the unknowns in the solution. In other words, the final solution is given by $\mathbf{P}\cdot\mathbf{x} = \mathbf{b}'$, where $\mathbf{b}'$ is the last column of the augmented matrix after the solution has been found.

A function such as *lu* in SCILAB automatically take care of using partial or full pivoting in the solution of linear systems. No special provision is necessary in the call to this function to activate pivoting. The function *lu* take care of reincorporating the permutation matrix into the final result.

Consider, as an example, the linear system shown below:

$$\begin{aligned} 2Y + 3Z &= 7, \\ 2X + 3Z &= 13, \\ 8X + 16Y - Z &= -3. \end{aligned}$$

The augmented matrix is:

$$\mathbf{A}_{aug} = \begin{bmatrix} 0 & 2 & 3 & 7 \\ 2 & 0 & 3 & 13 \\ 8 & 16 & -1 & -3 \end{bmatrix}.$$

The zero in element (1,1) will not allow the simple Gaussian elimination to proceed since a division by zero will be required. Trying a solution with function *gausselim*, described earlier, produces an error:

```
-->A = [0,2,3;2,0,3;8,16,-1], b = [7;13;-3]
 A  =

!   0.    2.     3. !
!   2.    0.     3. !
!   8.    16.  - 1. !
 b  =

!   7.  !
!   13. !
! - 3.  !

-->getf('gausselim')

-->x = gausselim(A,b)
 !--error    27
division by zero...
at line      26 of function gausselim               called by :
x = gausselim(A,b)
```

As expected, *gausselim* produces a division-by-zero error. However, *lu*, which already incorporates pivoting, will produce the following solution through LU decomposition:

```
-->[L,U,P] = lu(A)
 P  =

!   0.    0.    1. !
!   0.    1.    0. !
!   1.    0.    0. !
 U  =
```



```
!   8.    16.  - 1.    !
!   0.  - 4.     3.25  !
!   0.    0.     4.625 !
 L  =

!   1.      0.     0. !
!    .25    1.     0. !
!   0.    -  .5    1. !
```

The right-hand side vector is modified by using the permutation matrix:

```
-->c = P*b
 c  =

! - 3.  !
!   13. !
!   7.  !
```

The solution to the system of linear equations through LU decomposition proceeds in two parts:

```
-->y = L\c
 y  =

! - 3.     !
!   13.75  !
!   13.875 !

-->x = U\y
 x  =

!   2. !
! - 1. !
!   3. !
```

Gaussian elimination with partial pivoting

In this section we modify the function *gausselim* to incorporate partial pivoting. The resulting function is called *gausselimPP*. The algorithm for partial pivoting requires us to compare the current pivot at the *k-th* pass of the forward elimination part, namely a_{kk}, with all the elements of the same column, a_{ij}, for i>k. When the largest absolute value of the column is found, the corresponding row and row k are exchanged before proceeding with the forward elimination algorithm. The backward substitution part of the algorithm proceeds as in the original *gausselim* function.

Here is the listing of function *gausselimPP*:

```
function [x] = gausselimPP(A,b)

//This function obtains the solution to the system of
//linear equations A*x = b, given the matrix of coefficients A
//and the right-hand side vector, b.  Gaussian elimination with
//partial pivoting.
```



```
[nA,mA] = size(A)
[nb,mb] = size(b)

if nA<>mA then
       error('gausselim - Matrix A must be square');
       abort;
elseif mA<>nb then
        error('gausselim - incompatible dimensions between A and b');
       abort;
end;

a = [A b];    // Augmented matrix
n = nA   ;    // Matrix size

//Forward elimination with partial pivoting

for k=1:n-1
    kpivot = k; amax = abs(a(k,k));        //Pivoting
    for i=k+1:n
        if abs(a(i,k))>amax then
              kpivot = i; amax = a(k,i);
       end;
    end;
    temp = a(kpivot,:); a(kpivot,:) = a(k,:); a(k,:) = temp;

    for i=k+1:n                            //Forward elimination
       for j=k+1:n+1
              a(i,j)=a(i,j)-a(k,j)*a(i,k)/a(k,k);
       end;
    end;
end;

//Backward substitution

x(n) = a(n,n+1)/a(n,n);

for i = n-1:-1:1
       sumk=0
       for k=i+1:n
              sumk=sumk+a(i,k)*x(k);
       end;
       x(i)=(a(i,n+1)-sumk)/a(i,i);
end;

//End function
```

Application of the function *gausselimPP* for the case under consideration is shown next:

```
-->A = [0,2,3;2,0,3;8,16,-1]; b = [7;13;-3];

-->getf('gausselimPP')

-->gausselimPP(A,b)
 ans  =

!   2. !
! - 1. !
!   3. !
```


Solving multiple set of equations with the same coefficient matrix

Suppose that you want to solve the following three sets of equations:

$$\begin{array}{lll} x_1 + 2x_2 + 3x_3 = 14, & 2x_1 + 4x_2 + 6x_3 = 9, & 2x_1 + 4x_2 + 6x_3 = -2, \\ 3x_1 - 2x_2 + x_3 = 2, & 3x_1 - 2x_2 + x_3 = -5, & 3x_1 - 2x_2 + x_3 = 2, \\ 4x_1 + 2x_2 - x_3 = 5, & 4x_1 + 2x_2 - x_3 = 19, & 4x_1 + 2x_2 - x_3 = 12. \end{array}$$

We can write the three systems of equations as a single matrix equation: $\mathbf{A}\cdot\mathbf{X} = \mathbf{B}$, where

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 3 & -2 & 1 \\ 4 & 2 & -1 \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 14 & 9 & -2 \\ 2 & -5 & 2 \\ 5 & 19 & 12 \end{bmatrix}.$$

In the unknown matrix **X**, the first sub-index identifies the original sub-index, while the second one identifies to which system of linear equations a particular variable belongs.

The solution to the system, $\mathbf{A}\cdot\mathbf{X} = \mathbf{B}$, can be found in SCILAB using the backward slash operator, i.e., $\mathbf{X} = \mathbf{A}\backslash\mathbf{B}$, or the inverse matrix, $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$. For the matrices defined above we can write:

```
-->A = [1,2,3;3,-2,1;4,2,-1], B=[14,9,-2;2,-5,2;5,19,12]
 A  =
!   1.    2.    3. !
!   3.  - 2.    1. !
!   4.    2.  - 1. !
 B  =
!   14.    9.   - 2.  !
!   2.   - 5.     2.  !
!   5.     19.    12. !
```

Solution using the backward slash operator:

```
-->X1 = A\B
 X1  =

!   1.    2.    2. !
!   2.    5.    1. !
!   3.  - 1.  - 2. !
```

Solution using the inverse matrix of A:

```
-->X2 = inv(A)*B
 X2  =

!   1.    2.    2. !
!   2.    5.    1. !
!   3.  - 1.  - 2. !
```

In both cases, the result is:

$$x := \begin{bmatrix} 1 & 2 & 2 \\ 2 & 5 & 1 \\ 3 & -1 & -2 \end{bmatrix}.$$

Gaussian elimination for multiple sets of linear equations

In this section, function *gausselimPP* is modified to account for a multiple set of linear equations. The augmented matrix for this case is $\mathbf{A}_{aug}$ = [**A B**], where A is a nxn matrix, and B is a nxm matrix. The solution is now a nxn matrix **X**.

```
function [x] = gausselimm(A,B)

//This function obtains the solution to the system of
//linear equations A*X = B, given the nxn matrix of coefficients A
//and the nxm right-hand side matrix, B.  Matrix X is nxm.

[nA,mA] = size(A)
[nB,mB] = size(B)

if nA<>mA then
      error('gausselim - Matrix A must be square');
      abort;
elseif mA<>nB then
       error('gausselim - incompatible dimensions between A and b');
      abort;
end;

a = [A B];    // Augmented matrix
n = nA   ;    // Number of rows and columns in A, rows in B
m = mB   ;    // Number of columns in B

//Forward elimination with partial pivoting

for k=1:n-1
    kpivot = k; amax = abs(a(k,k));        //Pivoting
    for i=k+1:n
        if abs(a(i,k))>amax then
            kpivot = i; amax = a(k,i);
      end;
    end;
    temp = a(kpivot,:); a(kpivot,:) = a(k,:); a(k,:) = temp;

    for i=k+1:n                           //Forward elimination
      for j=k+1:n+m
            a(i,j)=a(i,j)-a(k,j)*a(i,k)/a(k,k);
      end;
    end;
end;

//Backward substitution

for j = 1:m
      x(n,j) = a(n,n+j)/a(n,n);
      for i = n-1:-1:1
            sumk=0
```



```
            for k=i+1:n
                  sumk=sumk+a(i,k)*x(k,j);
            end;
            x(i,j)=(a(i,n+j)-sumk)/a(i,i);
      end;
end;


//End function
```

Next, function *gausselimm* is applied to the matrices A and B defined earlier:

```
-->A = [1,2,3;3,-2,1;4,2,-1]; B=[14,9,-2;2,-5,2;5,19,12];

-->getf('gausselimm')

-->gausselimm(A,B)
 ans  =

!   1.    2.    2. !
!   2.    5.    1. !
!   3.  - 1.  - 2. !
```

Calculating an inverse matrix using Gaussian elimination

If the right-hand side matrix **B** of a set of multiple systems of linear equations is the nxn identity matrix, i.e., $\mathbf{B} = \mathbf{I}_{nxn}$, the solution **X** from the matrix system $\mathbf{A}\cdot\mathbf{X} = \mathbf{B}$, is the inverse of matrix **A**, i.e., $\mathbf{X} = \mathbf{A}^{-1}$.

For example, to find the inverse of the matrix **A** presented earlier through the use of function *gausselimm*, you can use the following SCILAB command:

```
-->Ainv = gausselimm(A,eye(3,3))
 Ainv  =

!   0.        .1428571     .1428571 !
!    .125  -  .2321429     .1428571 !
!    .25      .1071429  -  .1428571 !
```

You can check that this result is the same as that obtained from SCILAB's function *inv*:

```
-->inv(A)
 ans  =

!   0.        .1428571     .1428571 !
!    .125  -  .2321429     .1428571 !
!    .25      .1071429  -  .1428571 !
```

Calculating the determinant with Gaussian elimination

You can prove that the determinant of a matrix can be obtained by multiplying the elements in the main diagonal after the forward elimination part of a Gaussian elimination procedure is

completed. In other words, after completing the forward elimination of a "naïve" Gaussian elimination, the determinant of a nxn matrix **A** = [a_{ij}] is calculated as

$$\det(\mathbf{A}) = \prod_{k=1}^{n} a_{kk}.$$

If partial pivoting is included, however, the sign of the determinant changes according to the number of row switches included in the pivoting process. In such case, if N represents the number of row exchanges, the determinant is given by

$$\det(\mathbf{A}) = (-1)^N \cdot \prod_{k=1}^{n} a_{kk}.$$

The calculation of the determinant is included in the Gaussian elimination function called *gausselimd*. The call to this function includes return variables for the solution **X** to the system **A**·**X** = **B** as well as for the determinant of A. A listing of the function is shown below:

```
function [x,detA] = gausselimd(A,B)

//This function obtains the solution to the system of
//linear equations A*X = B, given the nxn matrix of coefficients A
//and the nxm right-hand side matrix, B.  Matrix X is nxm.

[nA,mA] = size(A);
[nB,mB] = size(B);

if nA<>mA then
       error('gausselim - Matrix A must be square');
       abort;
elseif mA<>nB then
        error('gausselim - incompatible dimensions between A and b');
       abort;
end;
a = [A B];    // Augmented matrix
n = nA   ;    // Number of rows and columns in A, rows in B
m = mB   ;    // Number of columns in B

//Forward elimination with partial pivoting
nswitch = 0;
for k=1:n-1
    kpivot = k; amax = abs(a(k,k));        //Pivoting
    for i=k+1:n
        if abs(a(i,k))>amax then
             kpivot = i; amax = a(k,i);
             nswitch = nswitch+1
       end;
    end;
    temp = a(kpivot,:); a(kpivot,:) = a(k,:); a(k,:) = temp;

    for i=k+1:n                            //Forward elimination
       for j=k+1:n+m
             a(i,j)=a(i,j)-a(k,j)*a(i,k)/a(k,k);
       end;
    end;
end;
```



```
//Calculating the determinant

detA = 1;
for k = 1:n
       detA = detA*a(k,k);
end;
detB = (-1)^nswitch*detA;

//Checking for singular matrix
epsilon = 1.0e-10;
if abs(detB)<epsilon then
       error('gausselimd - singular matrix');
       abort;
end;

//Backward substitution

for j = 1:m
       x(n,j) = a(n,n+j)/a(n,n);
       for i = n-1:-1:1
              sumk=0
              for k=i+1:n
                     sumk=sumk+a(i,k)*x(k,j);
              end;
              x(i,j)=(a(i,n+j)-sumk)/a(i,i);
       end;
end;

//End function
```

An application of function *gausselimd* for a non-singular matrix A is shown next:

```
-->A = [3,5,-1;2,2,3;1,1,2]
 A  =

!   3.    5.  - 1. !
!   2.    2.    3. !
!   1.    1.    2. !

-->b = [-4;17;11]
 b  =

! - 4.  !
!   17. !
!   11. !


-->getf('gausselimd')

-->[x,detA] = gausselimd(A,b)
 detA  =

  - 2.
 x  =

!   2. !
! - 1. !
!   5. !
```


For a singular matrix A, function *gausselimd* provides an error message:

```
-->A = [2,3,1;4,6,2;1,1,2]
 A  =

!   2.    3.    1. !
!   4.    6.    2. !
!   1.    1.    2. !


-->[x,detA] = gausselimd(A,b)
 !--error  9999
gausselimd - singular matrix
at line      53 of function gausselimd              called by :
[x,detA] = gausselimd(A,b)
```

Gauss-Jordan elimination

Gauss-Jordan elimination consists in continuing the row operations in the augmented matrix that results from the forward elimination from a Gaussian elimination process, until an identity matrix is obtained in place of the original **A** matrix. For example, for the following augmented matrix, the forward elimination procedure results in:

$$\mathbf{A}_{aug} = \left(\begin{array}{ccc|c} 2 & 4 & 6 & 14 \\ 3 & -2 & 1 & -3 \\ 4 & 2 & -1 & -4 \end{array}\right) \cong \left(\begin{array}{ccc|c} 2 & 4 & 6 & 14 \\ 0 & -8 & -8 & -24 \\ 0 & -6 & -13 & -32 \end{array}\right) \cong \left(\begin{array}{ccc|c} 2 & 4 & 6 & 14 \\ 0 & -8 & -8 & -24 \\ 0 & 0 & -7 & -14 \end{array}\right)$$

We can continue performing row operations until the augmented matrix gets reduced to:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 0 & 1 & 1 & 3 \\ 0 & 0 & 1 & 2 \end{array}\right) \cong \left(\begin{array}{ccc|c} 1 & 2 & 3 & 7 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 2 \end{array}\right) \cong \left(\begin{array}{ccc|c} 1 & 2 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{array}\right) \cong \left(\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{array}\right)$$

The first matrix is the same as that one obtained before at the end of the forward elimination process, with the exception that all rows have been divided by the corresponding diagonal term, i.e., row 1 was divided by 2, row 2 was divided by -8, and row 3 was divided by –7. The final matrix is equivalent to the equations: X = -1, Y = 1, Z= 2, which is the solution to the original system of equations.

SCILAB provides the function *rref* (*r*ow-*r*educed *e*chelon *f*orm) that can be used to obtain a solution to a system of linear equations using Gauss-Jordan elimination. The function requires as argument an augmented matrix, for example:

```
-->A = [2,4,6;3,-2,1;4,2,-1]; b = [14;-3;-4];

-->A_aug = [A b]
 A_aug  =

!   2.    4.    6.    14. !
!   3.  - 2.    1.  - 3.  !
!   4.    2.  - 1.  - 4.  !

-->rref(A_aug)
 ans  =

!   1.    0.    0.  - 1. !
!   0.    1.    0.    1. !
!   0.    0.    1.    2. !
```

The last result indicates that the solution to the system is $x_1 = -1$, $x_2 = 1$, and $x_3 = 2$.

Calculating the inverse through Gauss-Jordan elimination

The calculation of an inverse matrix is equivalent to calculating the solution to the augmented system [**A** | **I**]. For example, for the matrix **A**

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

we would write this augmented matrix as

$$\mathbf{A}_{aug} = \left[\begin{array}{ccc|ccc} 1 & 2 & 3 & 1 & 0 & 0 \\ 3 & -2 & 1 & 0 & 1 & 0 \\ 4 & 2 & -1 & 0 & 0 & 1 \end{array}\right].$$

The SCILAB commands needed to obtain the inverse using Gauss-Jordan elimination are:

```
-->A = [1,2,3;3,-2,1;4,2,-1]  //Original matrix
 A  =
!   1.    2.    3. !
!   3.  - 2.    1. !
!   4.    2.  - 1. !

-->A_aug = [A eye(3,3)]      //Augmented matrix
 A_aug  =
!   1.    2.    3.    1.    0.    0. !
!   3.  - 2.    1.    0.    1.    0. !
!   4.    2.  - 1.    0.    0.    1. !
```


The next command produces the Gauss-Jordan elimination:

```
-->A_aug_inv = rref(A_aug)
 A_aug_inv  =
!   1.    0.    0.    0.        .1428571     .1428571 !
!   0.    1.    0.     .125  -   .2321429     .1428571 !
!   0.    0.    1.     .25      .1071429  -   .1428571 !
```

The inverse is obtained by extracting rows 4, 5, and 6, of the previous result:

```
-->A_inv_1 = [A_aug_inv(:,4), A_aug_inv(:,5), A_aug_inv(:,6)]
 A_inv_1  =

!   0.        .1428571     .1428571 !
!    .125  -  .2321429     .1428571 !
!    .25      .1071429  -  .1428571 !
```

To check that the matrix A_inv_1 is indeed the inverse of A use:

```
-->A*A_inv_1
 ans  =

!   1.    0.    0. !
!   0.    1.    0. !
!   0.    0.    1. !
```

This exercise is presented to illustrate the calculation of inverse matrices through Gauss-Jordan elimination. In practice, you should use the function *inv* in SCILAB to obtain inverse matrices. For the case under consideration, for example, the inverse is obtained from:

```
-->A_inv_2 = inv(A)
 A_inv_2  =

!   0.        .1428571     .1428571 !
!    .125  -  .2321429     .1428571 !
!    .25      .1071429  -  .1428571 !
```

The result is exactly the same as found above using Gauss-Jordan elimination.

Eigenvalues and eigenvectors

Given a square matrix **A**, we can write the *eigenvalue equation*

$$\mathbf{A}\cdot\mathbf{x} = \lambda\cdot\mathbf{x},$$

where the values of λ that satisfy the equation are known as the *eigenvalues of matrix **A***. For each value of λ, we can find, from the same equation, values of **x** that satisfy the eigenvalue equation. These values of **x** are known as the *eigenvectors of matrix **A***.

The eigenvalues equation can be written also as

$$(\mathbf{A} - \lambda\cdot\mathbf{I})\mathbf{x} = 0.$$

This equation will have a non-trivial solution only if the *characteristic matrix*, $(\mathbf{A} - \lambda\cdot\mathbf{I})$, is singular, i.e., if

$$\det(\mathbf{A} - \lambda\cdot\mathbf{I}) = 0.$$

The last equation generates an algebraic equation involving a polynomial of order *n* for a square matrix $\mathbf{A}_{n\times n}$. The resulting equation is known as the *characteristic polynomial* of matrix **A**. Solving the characteristic polynomial produces the *eigenvalues* of the matrix.

Using SCILAB we can obtain the characteristic matrix, characteristic polynomial, and eigenvalues of a matrix as shown below for a symmetric matrix

$$\mathbf{A} = \begin{bmatrix} 3 & -2 & 5 \\ -2 & 3 & 6 \\ 5 & 6 & 4 \end{bmatrix}.$$

First, we define the matrix A:

```
-->A = [3,-2,5;-2,3,6;5,6,4]
 A  =

!   3.  - 2.    5. !
! - 2.    3.    6. !
!   5.    6.    4. !
```

Next, the characteristic matrix **B** = $(\mathbf{A} - \lambda\cdot\mathbf{I})$ corresponding to a square matrix **A** is calculated as follows. Notice that we define the symbolic variable 'lam' (for lambda, λ) before forming the characteristic matrix:

```
-->lam = poly(0,'lam')
 lam  =

    lam

-->charMat = A-lam*eye(3,3)


 charMat  =

!   3 - lam   - 2           5         !
!                                     !
! - 2           3 - lam     6         !
!                                     !
!   5           6           4 - lam   !
```

The characteristic equation can be determined using function *poly* with matrix A and the variable name 'lam' as arguments, i.e.,

```
-->charPoly = poly(A,'lam')
 charPoly  =
```



```
                      2     3
    283 - 32lam -10lam + lam
```

The function *spec* produces the eigenvalues for matrix A:

```
-->lam = spec(A)
 lam  =

! - 5.4409348 !
!   4.9650189 !
!   10.475916 !
```

Calculating the first eigenvector

We can form the characteristic matrix for a particular eigenvalue, say λ_1 = *lam(1)*, by using:

```
-->B1 = A - lam(1)*eye(3,3)

B1 =

!  8.4409348  - 2.        5.      !
! - 2.         8.4409348   6.      !
!  5.         6.         9.4409348 !
```

This, of course, is a singular matrix, which you can check by calculating the determinant of B1:

```
--> det(B1)
ans   =

5.973E-14
```

which, although not exactly zero, is close enough to zero to ensure singularity.

Because the characteristic matrix is singular, there is no unique solution to the problem ($\mathbf{A}-\lambda_1\mathbf{I})\mathbf{x} = 0$. The equivalent system of linear equations is:

$$\begin{aligned} 8.4409348x_1 - 2x_2 + 5x_3 &= 0 \\ -2x_1 + 8.4409348x_2 + 6x_3 &= 0 \\ 5x_1 + 6x_2 + 9.4409348x_3 &= 0 \end{aligned}$$

The three equations are linearly dependent. This means that we can select an arbitrary value of one of the solutions, say, $x_3 = 1$, and solve two of the equations for the other two solutions, x_1 and x_2. We could try, for example, to solve the first two equations (with $x_3 = 1$), re-written as:

$$\begin{aligned} 8.4409348x_1 - 2x_2 &= -5 \\ -2x_1 + 8.4409348x_2 &= -6 \end{aligned}$$

Using SCILAB we can get the solution to this system as follows:

```
-->C1 = B1(1:2,1:2), b1 = -B1(3,1:2)
 C1  =

!   8.4409348  - 2.        !
```



```
! - 2.           8.4409348 !

-->b1 = -B1(1:2,3)
 b1  =

! - 5. !
! - 6. !
```

The solution for $\xi_1 = [x_1,x_2]^T$ is:

```
-->xi1 = C1\b1
 xi1  =

! -  .8060249 !
! -  .9018017 !
```

The eigenvector $\mathbf{x}_1$ is, therefore:

```
-->x1 = [xi1;1]
 x1  =

! -  .8060249 !
! -  .9018017 !
!   1.        !
```

If we want to find the corresponding unit eigenvector, $\mathbf{e}_{x1}$, we can use:

```
-->e_x1 = x1/norm(x1)
 e_x1 =

! - .5135977 !
! - .5746266 !
!   .6371983 !
```

Calculating eigenvectors with a user-defined function

Following the procedure outlined above for calculating the first eigenvector, we can write a SCILAB user-defined function to calculate all the eigenvectors of a matrix. The following is a listing of this function:

```
function [x,lam] = eigenvectors(A)

//Calculates unit eigenvectors of matrix A
//returning a matrix x whose columns are
//the eigenvectors.  The function also
//returns the eigenvalues of the matrix.
[n,m] = size(A);
if m<>n then
       error('eigenvectors - matrix A is not square');
       abort;
end;

lam = spec(A)';                  //Eigenvalues of matrix A
```



```
x = [];

for k = 1:n
       B = A - lam(k)*eye(n,n);   //Characteristic matrix
        C = B(1:n-1,1:n-1);       //Coeff. matrix for reduced system
       b = -B(1:n-1,n);           //RHS vector for reduced system
       y = C\b;                   //Solution for reduced system
       y = [y;1];                 //Complete eigenvector
       y = y/norm(y);             //Make unit eigenvector
       x = [x y];                 //Add eigenvector to matrix
end;

//End of function
```

Applying function *eigenvectors* to the matrix A used earlier produces the following eigenvalues and eigenvectors:

```
-->getf('eigenvectors')

-->[x,lam] = eigenvectors(A)
 lam  =

! - 5.4409348    4.9650189    10.475916 !
 x  =

! -  .5135977     .7711676     .3761887 !
! -  .5746266  -  .6347298     .5166454 !
!    .6371983     .0491799     .7691291 !
```

We can separate the eigenvectors $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ from matrix x by using:

```
-->x1 = x(1:3,1), x2 = x(1:3,2), x3 = x(1:3,3)
 x1  =

! -  .5135977 !
! -  .5746266 !
!    .6371983 !
 x2  =

!    .7711676 !
! -  .6347298 !
!    .0491799 !
 x3  =

!    .3761887 !
!    .5166454 !
!    .7691291 !
```

The eigenvectors corresponding to a symmetric matrix are orthogonal to each other, i.e., $\mathbf{x}_i \bullet \mathbf{x}_j = 0$, for $i \neq j$. Checking these results for the eigenvectors found above:

```
-->x1'*x2, x1'*x3, x2'*x3
 ans  =

  - 3.678E-16
 ans  =
```



```
  - 5.551E-17
 ans  =

    9.576E-16
```

For a non-symmetric matrix, the eigenvalues may include complex numbers:

Example 1 - non-symmetric matrix, real eigenvalues

```
-->A = [3, 2, -1; 1, 1, 2; 5, 5, -2]    A  =

!   3.    2.  - 1. !
!   1.    1.    2. !
!   5.    5.  - 2. !

-->[x,lam] = eigenvectors(A)

 lam  =

!    .9234927    4.1829571  - 3.1064499 !
 x  =

! -  .5974148     .3437456     .2916085 !
!    .7551776     .5746662  -  .4752998 !
!    .2698191     .7426962     .8300931 !
```

Example 2 - non-symmetric matrix, complex eigenvalues

```
-->A = [2, 2, -3; 3, 3, -2; 1, 1, 1]
A  =

!   2.    2.  - 3. !
!   3.    3.  - 2. !
!   1.    1.    1. !

-->[x,lam] = eigenvectors(A)
 warning
 matrix is close to singular or badly scaled.
 results may be inaccurate. rcond =   7.6862E-17

 lam  =

!   4.646E-16    3. - i      3. + i   !
 x  =

! -  .7071068     .25 -  .25i     .25 +  .25i !
!    .7071068     .75 -  .25i     .75 +  .25i !
!   2.826E-16     .5              .5          !
```

Generating the characteristic equation for a matrix

This section represents an additional exercise on SCILAB function programming, since we already know that SCILAB can generate the characteristic equation for a square matrix through function *poly*. The method coded in the function uses the equations:

$$p_{n+1} = -1.0,\ \mathbf{B}_{n+1} = \mathbf{A}$$

$$\mathbf{B}_j = \mathbf{A}\cdot(\mathbf{B}_{j+1} - p_{j+1}\mathbf{I}),\ \text{for } j = n, n-1, \ldots, 3, 2, 1$$

$$P_j = (-1)^j p_j,\ j = 1, 2, \ldots, n+1$$

The matrix **I** shown in the formula is the nxn identity matrix. The coefficients of the polynomial will be contained in the 1x(n+1) vector

$$p = [p_1\ p_2\ \ldots\ p_{n+1}].$$

The characteristic equation is given by

$$p_1 + p_2\lambda + p_3\lambda^2 + \ldots + p_n\lambda^{n-1} + p_{n+1}\lambda^n = 0.$$

Here is a listing of the function:

```
function [p]=chreq(A)
//This function generates the coefficients of the characteristic
//equation for a square matrix A
[m n]=size(A);
if(m <> n)then
       error('matrix is not square.')
       abort
end;
I = eye(n,n);                    //Identity matrix
p = zeros(1,n);                  //Matrix (1xn) filled with zeroes
p(1,n+1) = -1.0;
B = A;
p(1,n)=trace(B);
for j = n-1:-1:1,
       B = A*(B-p(1,j+1)*I),
       p(1,j) = trace(B)/(n-j+1),
end;
p = (-1)^n*p;
p = poly(p,"lmbd","coeff");
p
//end function chreq
```

In this function we used the function *poly* to generate the final result in the function. In the call to *poly* shown above, we use the vector of coefficients p as the first argument, "lmbd" is the independent variable λ, and "coeff" is required to indicate that the vector p represent the coefficients of the polynomial. The following commands show you how to load the function and run it for a particular matrix:

```
-->getf('chreq')

-->A = [1 3 1;2 5 -1;2 7 -1]
 A  =

!   1.    3.    1. !
```



```
!   2.    5.  - 1. !
!   2.    7.  - 1. !

-->CheqA = chreq(A)
 CheqA  =

                      2      3
  - 6 - 2lmbd - 5lmbd + lmbd
```

Generalized eigenvalue problem

The *generalized eigenvalue problem* involves square matrices **A** and **B** of the same size, and consists in finding the values of the scalar λ and the vectors **x** that satisfy the matricial equation:

$$\mathbf{A}\cdot\mathbf{x} = \lambda\cdot\mathbf{B}\cdot\mathbf{x}.$$

The following function, *geingenvectors* (*g*eneralized *eigenvectors*), calculates the eigenvalues and eigenvectors from the generalized eigenvalue problem $\mathbf{A}\cdot\mathbf{x} = \lambda\cdot\mathbf{B}\cdot\mathbf{x}$, or $(\mathbf{A}-\lambda\mathbf{B})\mathbf{x} = 0$.

```
function [x,lam] = geigenvectors(A,B)

//Calculates unit eigenvectors of matrix A
//returning a matrix x whose columns are
//the eigenvectors.  The function also
//returns the eigenvalues of the matrix.

[nA,mA] = size(A);
[nB,mB] = size(B);

if (mA<>nA | mB<>nB) then
       error('geigenvectors - matrix A or B not square');
       abort;
end;

if nA<>nB then
       error('geigenvectors - matrix A and B have different dimensions');
       abort;
end;

lam = poly(0,'lam');               //Define variable "lam"
chPoly = det(A-B*lam);             //Characteristic polynomial
lam = roots(chPoly)';              //Eigenvalues of matrix A

x = []; n = nA;

for k = 1:n
       BB = A - lam(k)*B;          //Characteristic matrix
        CC = BB(1:n-1,1:n-1);      //Coeff. matrix for reduced system
       bb = -BB(1:n-1,n);          //RHS vector for reduced system
       y = CC\bb;                  //Solution for reduced system
       y = [y;1];                  //Complete eigenvector
       y = y/norm(y);              //Make unit eigenvector
       x = [x y];                  //Add eigenvector to matrix
end;
```



```
//End of function
```

An application of this function is presented next:

```
-->A = [4,1,6;8,5,8;1,5,5]
 A  =

!   4.    1.    6. !
!   8.    5.    8. !
!   1.    5.    5. !

-->B = [3,9,7;3,3,2;9,3,4]
 B  =

!   3.    9.    7. !
!   3.    3.    2. !
!   9.    3.    4. !

-->getf('geigenvectors')

-->[x,lam] = geigenvectors(A,B)
 lam  =

! -  .2055713 - 1.1759636i  -  .2055713 + 1.1759636i  - 1.5333019 !
 x  =

! -  .2828249 -  .0422024i  -  .2828249 +  .0422024i  -  .0202307 !
! -  .5392352 +  .2691247i  -  .5392352 -  .2691247i  -  .7437197 !
!    .7450009                .7450009                   .6681854 !
```


Sparse matrices

Sparse matrices are those that have a large percentage of zero elements. When a matrix is defined as sparse in SCILAB, only those non-zero elements are stored. The regular definition of a matrix, also referred to as a *full* matrix, implies that SCILAB stores all elements of the matrix, zero or otherwise.

Creating sparse matrices

SCILAB provides the function *sparse* to define sparse matrices. The simplest call to the function is

A_sparse = sparse(A)

Where A is a sparse or full matrix. Obviously, if A is already a sparse matrix, no action is taken by the call to function *sparse.* If A is a full matrix, the function *sparse* squeezes out those zero elements from A. For example,

```
-->A = [2, 0, -1; 0, 1, 0; 0, 0 , 2]
 A  =

!   2.    0.  - 1. !
!   0.    1.    0. !
!   0.    0.    2. !

-->As = sparse(A)
 As  =

(    3,    3) sparse matrix

(    1,    1)        2.
(    1,    3)      - 1.
(    2,    2)        1.
(    3,    3)        2.
```

Notice that SCILAB reports the size of the matrix (3,3), and those non-zero elements only. These are the only elements stored in memory. Thus, sparse matrices are useful in minimizing memory storage particularly when large-size matrices, with relatively small density of non-zero elements, are involved.

Alternatively, a call to *sparse* of the form

A_sparse = sparse(index,values)

Where *index* is a nx2 matrix and *values* is a nx1 vector such that *values(i,1)* represents the element in row *index(i,1)* and column *index(i,2)* of the sparse matrix. For example,

```
-->row = [2, 2, 3, 3, 6, 6, 10]  //row position for non-zero elements
 row  =

!   2.    2.    3.    3.    6.    6.    10. !

-->col = [1, 2, 2, 3, 1, 4, 2]   //column position for non-zero elements
 col  =
```



```
!   1.    2.    2.    3.    1.    4.    2. !

-->val = [-0.5, 0.3, 0.2, 1.5, 4.2, -1.1, 2.0] //values of non-zero elements
 val  =

! -  .5     .3     .2    1.5    4.2  - 1.1    2. !


-->index = [row' col']    //indices of non-zero elements (i,j)
 index  =

!   2.     1. !
!   2.     2. !
!   3.     2. !
!   3.     3. !
!   6.     1. !
!   6.     4. !
!   10.    2. !

-->As = sparse(index,val)   //creating the sparse matrix
 As  =

(   10,    4) sparse matrix

(    2,    1)      -  .5
(    2,    2)         .3
(    3,    2)         .2
(    3,    3)        1.5
(    6,    1)        4.2
(    6,    4)      - 1.1
(   10,    2)        2.
```

The function *full* converts a sparse matrix into a full matrix. For example,

```
-->A =full(As)
 A  =

!   0.     0.     0.     0.  !
! -  .5     .3    0.     0.  !
!   0.      .2    1.5    0.  !
!   0.     0.     0.     0.  !
!   0.     0.     0.     0.  !
!   4.2    0.     0.   - 1.1 !
!   0.     0.     0.     0.  !
!   0.     0.     0.     0.  !
!   0.     0.     0.     0.  !
!   0.     2.     0.     0.  !
```

The following call to *sparse* includes defining the row and column dimensions of the sparse matrix besides providing indices and values of the non-zero values:

A_sparse = sparse(index,values,dim)

Here, *dim* is a 1x2 vector with the row and column dimensions of the sparse matrix. As an example, we can try:

```
-->As = sparse(index,val,[10,12])
```



```
 As  =

(   10,   12) sparse matrix

(    2,    1)      -  .5
(    2,    2)         .3
(    3,    2)         .2
(    3,    3)        1.5
(    6,    1)        4.2
(    6,    4)      - 1.1
(   10,    2)        2.
```

Getting information about a sparse matrix

The function *spget,* through a call of the form

[index, val, dim] = spget(As)

returns the location of the values contained in *val* described by the row and column indices (listed in the first and second columns of *index*), as well as the row and column dimensions of the matrix contained in vector *dim.* For example, for the matrix *As* defined above, function *spget* produces:

```
-->[index, values, dim] = spget(As)
 dim  =

!   10.    12. !
 values  =

! -  .5 !
!    .3 !
!    .2 !
!   1.5 !
!   4.2 !
! - 1.1 !
!   2.  !
 index  =

!   2.     1. !
!   2.     2. !
!   3.     2. !
!   3.     3. !
!   6.     1. !
!   6.     4. !
!   10.    2. !
```

Sparse matrix with unit entries

To create a matrix including values of 1.0 with the same structure of an existing sparse matrix use the function *spones*, for example:


```
-->A1 = spones(As)
 A1  =

(   10,   12) sparse matrix

(    2,    1)        1.
(    2,    2)        1.
(    3,    2)        1.
(    3,    3)        1.
(    6,    1)        1.
(    6,    4)        1.
(   10,    2)        1.
```

Sparse identity matrices

To create a sparse identity matrix use the function *speye*. For example, giving the dimensions of the desired identity matrix we can use:

```
-->speye(6,6)
 ans  =

(    6,    6) sparse matrix

(    1,    1)        1.
(    2,    2)        1.
(    3,    3)        1.
(    4,    4)        1.
(    5,    5)        1.
(    6,    6)        1.
```

To produce a sparse identity matrix with the same dimensions of an existing matrix *As* use:

```
-->speye(As)
 ans  =

(   10,   12) sparse matrix

(    1,    1)        1.
(    2,    2)        1.
(    3,    3)        1.
(    4,    4)        1.
(    5,    5)        1.
(    6,    6)        1.
(    7,    7)        1.
(    8,    8)        1.
(    9,    9)        1.
(   10,   10)        1.
```

Sparse matrix with random entries

The function *sprand* generates a sparse matrix with random non-zero entries. The function requires the dimensions of the matrix as the two first arguments, and a density of non-zero

elements as the third argument. The density of non-zero values is given as a number between zero and one, for example:

```
-->As = sprand(6,5,0.2)
 As  =

(    6,    5) sparse matrix

(    1,    3)         .0500420
(    2,    1)         .9931210
(    2,    4)         .7485507
(    3,    2)         .6488563
(    3,    5)         .4104059
(    4,    2)         .9923191
```

To see the full matrix use:

```
-->full(As)
 ans  =

!   0.           0.            .0500420    0.           0.         !
!    .9931210    0.           0.            .7485507    0.         !
!   0.            .6488563    0.           0.            .4104059 !
!   0.            .9923191    0.           0.           0.         !
!   0.           0.           0.           0.           0.         !
!   0.           0.           0.           0.           0.         !
```

Sparse matrices with zero entries

The function *spzeros* is used to define a sparse matrix given the dimensions of the matrix. Since all the elements of the matrix are zero, what the function does is to reserve memory space for the matrix ensuring its sparse character. A couple of examples of application of *spzeros* are shown next:

```
->spzeros(As)
 ans  =
(    6,    5) zero sparse matrix

-->spzeros(6,4)
 ans  =
(    6,    4) zero sparse matrix
```

The function *spzeros* can be used, for example, in programming SCILAB functions that require sparse matrices to reserve a matrix name for future use.

Visualizing sparse matrices

The following function, *spplot* (*sp*arse matrix *plot*), can be used to visualize the distribution of non-zero elements for relatively large sparse matrices. A listing of the function follows:

```
function spplot(As)

//Plot a schematic of non-zero elements
//for a sparse matrix As
```



```
wnumber = input('Enter graphics window number:');
A = sparse(As);
[index,vals,dim] = spget(A);
xx=index(:,1);yy=index(:,2);
xset('window',wnumber);
xset('mark',-2,2);
plot2d(xx,yy,-2);

//end function
```

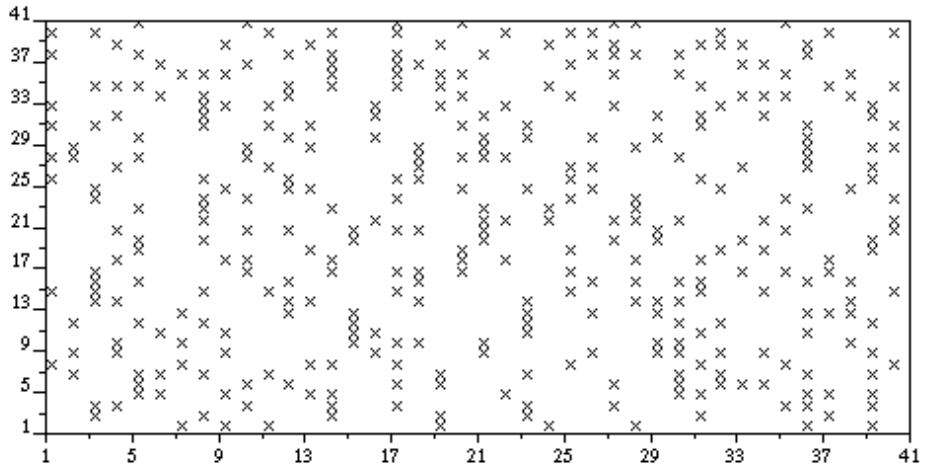
The function requires as input the name of the sparse matrix and it requests the number of the graphics window where the plot is to be displayed. The following is an example that uses this function to visualize a sparse matrix of dimensions 40x40 with a density of non-zero numbers of 0.2:

```
-->As = sprand(40,40,0.2);
-->getf('spplot')
-->spplot(As)

 Enter graphics window number:

--> 2
```

The result, shown in SCILAB graphics window number 2 is reproduced next:



Factorization of sparse matrices

SCILAB provides functions *lufact*, *luget*, and *ludel*, to produce the LU factorization of a sparse matrix A. Function *lufact* returns the rank of the matrix as well as a pointer or handle to the LU factors, which are then retrieved with function *luget*. Function *ludel* is required after performing a LU factorization to clear up the pointer or handle generated with *lufact*.

The call to function *lufact* has the general form:

[hand,rk] = lufact(As)

where *hand* is the handle or pointer, *rk* is the rank of sparse matrix *As*. Be aware that *hand* is a pointer to locate the LU factors in memory. SCILAB produces no display for *hand*.

The call to function *luget* has the general form:

[P,L,U,Q] = luget(hand)

where P and Q are permutation matrices and L and U are the LU factors of the sparse matrix that generated *hand* through a call to function *lufact*. Matrices P, Q, L, and U are related to matrix As by P*Q*L*U = As.

After a LU factorization is completed, it is necessary to call function *ludel* to clear the pointer generated in the call to function *lufact*, i.e., use

ludel(hand)

The following example shows the LU factorization of a 6x6 randomly-generated sparse matrix with a density of non-zero numbers of 0.5:

```
-->As = sprand(5,5,0.5);
```

To see the full matrix just generated try:

```
-->full(As)
 ans  =

!    .7019967     .7354560    0.           0.            .5098139 !
!    .5018194    0.            .6522234     .2921187     .3776263 !
!   0.            .4732295     .9388094     .6533222    0.        !
!    .7860680    0.           0.            .9933566    0.        !
!   0.            .9456872     .2401141     .4494063    0.        !
```

The first step in the LU factorization is:

```
-->[hand,rk] = lufact(As)
 rk  =

    5.
 hand  =
```

The rank of the matrix is 5. Notice that SCILAB shows nothing for the handle or pointer *hand*. The second step in the LU factorization is to get the matrices P, L, U, and Q, such that P*L*U*Q = As, i.e.,

```
-->[P,L,U,Q] = luget(hand);
```

These matrices are shown in full as follows:

```
-->full(P)
 ans  =

!   0.    0.    1.    0.    0. !
!   0.    0.    0.    1.    0. !
```



```
!   0.    1.    0.    0.    0. !
!   1.    0.    0.    0.    0. !
!   0.    0.    0.    0.    1. !

-->full(L)
 ans  =

!    .9933566    0.           0.           0.           0.        !
!    .6533222     .9388094    0.           0.           0.        !
!   0.           0.            .7019967    0.           0.        !
!    .2921187     .6522234     .6298297  -  .9886182    0.        !
!    .4494063     .2401141  -  .2233988    1.0586985     .0768072 !

-->full(U)
 ans  =

!   1.    0.     .7913251    0.           0.        !
!   0.    1.  -  .5506871     .5040741    0.        !
!   0.    0.    1.           1.047663     .726234  !
!   0.    0.    0.           1.            .0806958 !
!   0.    0.    0.           0.           1.        !

-->full(Q)
 ans  =

!   0.    0.    0.    1.    0. !
!   0.    0.    1.    0.    0. !
!   1.    0.    0.    0.    0. !
!   0.    1.    0.    0.    0. !
!   0.    0.    0.    0.    1. !
```

To check if the product P*L*U*Q is indeed equal to the original sparse matrix As, use:

```
-->full(P*L*U*Q-As)
 ans  =

!   0.    1.110E-16    0.    0.    0. !
!   0.    0.           0.    0.    0. !
!   0.    0.           0.    0.    0. !
!   0.    0.           0.    0.    0. !
!   0.    1.110E-16    0.    0.    0. !
```

The resulting matrix has a couple of small non-zero elements. These can be cleared by using the function *clean* as follows:

```
-->clean(full(P*L*U*Q-As))
 ans  =

!   0.    0.    0.    0.    0. !
!   0.    0.    0.    0.    0. !
!   0.    0.    0.    0.    0. !
!   0.    0.    0.    0.    0. !
!   0.    0.    0.    0.    0. !
```

At this point we can use function *ludel* to clear up the handle or pointer used in the factorization:

```
--> ludel(hand)
```


Solution to system of linear equations involving sparse matrices

The solution to a system of linear equations of the form **A**·**x** = **b**, where **A** is a sparse matrix and **b** is a full column vector, can be accomplished by using functions *lufact* and *lusolve*. Function *lufact* provides the handle or pointer *hand*, from [*hand,rk*] = lufact(*A*), which then used by *lusolve* through the function call

x = lusolve(hand,b)

The following example illustrates the uses of *lufact* and *lusolve* to solve a system of linear equations whose matrix of coefficients is sparse:

```
-->A=sprand(6,6,0.6);       //Generate a random, sparse matrix of coefficients

-->b=rand(6,1);             //Generate a random, full right-hand side vector
```

First, we get the rank and handle for the LU factorization of matrix A:

```
-->[hand,rk] = lufact(A)
 rk  =

    5.
 hand  =
```

Next, we use function *lusolve* with the handle *hand* and right-hand side vector *b* to obtain the solution to the system of linear equations:

```
-->x = lusolve(hand,b)
 x  =

! -  .8137973 !
! -  .5726203 !
!    .8288245 !
!    .9152823 !
!    .4984395 !
!   0.        !
```

Function *lusolve* also allows for the direct solution of the system of linear equations without having to use *lufact* first. For the case under consideration the SCILAB command is:

```
-->x = lusolve(A,b)
                !--error    19
singular matrix
```

However, matrix A for this case is singular (earlier we found that its rank was 5, i.e., smaller than the number of rows or columns, thus indicating a singular matrix), and no solution is obtained. The use of *lufact* combined with *lusolve*, as shown earlier, forces a solution by making one of the values equal to zero and solving for the remaining five values.

After completing the solution we need to clear the handle for the LU factorization by using:

```
-->ludel(hand)
```


A second example of solving a system of linear equations is shown next:

```
-->A = sprand(5,5,0.5);
-->b = full(sprand(5,1,0.6));
```

Notice that to generate b, which must be a full vector, we first generated a sparse random vector with a non-zero value density of 0.6. Next, we used the function *full* to convert that sparse vector into a full one.

A call to function *lufact* will provide the rank of matrix A:

```
-->[hand,rk] = lufact(A)
 rk  =

    5.
 hand  =
```

Because the matrix has full rank (i.e., its rank is the same as the number of rows or columns), it is possible to find the solution by using:

```
-->x = lusolve(A,b)
 x  =

! -  .2813594 !
!    .5590971 !
!   1.0143263 !
! -  .1810458 !
! - 1.1233045 !
```

Do not forget to clear the handle generated with *lufact* by using: `-->ludel(hand)`

Solution to system of linear equations using the inverse of a sparse matrix

If sparse matrix **A** in the system of linear equations $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$ is non-singular (i.e., a full-rank matrix), it is possible to find the inverse of **A**, $\mathbf{A}^{-1}$, through function *inv.* The solution x can then be found with $\mathbf{x} = \mathbf{A}^{-1}\cdot\mathbf{b}$. For example, using matrix A from the previous example we can write:

```
-->Ainv = inv(A)                //Calculate inverse of a sparse matrix
 Ainv  =

(    5,    5) sparse matrix

(    1,    1)      -  .7235355
(    1,    2)      -  .7404212
(    1,    3)         .3915968
(    1,    4)        2.0631067
(    1,    5)         .1957261
(    2,    1)        1.4075482
(    2,    2)        1.2851264
(    2,    3)      -  .6179589
(    2,    4)         .0607043
(    2,    5)      -  .5012662
(    3,    1)        2.9506345
(    4,    1)      - 2.1683689
```



```
(    4,    3)        1.8737551
(    5,    1)      - 2.4138658
(    5,    2)      -  .0298745
(    5,    3)      -  .9542841
(    5,    4)      - 2.1488971
(    5,    5)        2.5469151

-->x = Ainv*b
 x  =

! -  .2813594 !
!    .5590971 !
!   1.0143263 !
! -  .1810458 !
! - 1.1233045 !
```

Alternatively, you can also use left-division to obtain the solution:

```
-->x=A\b
 x  =

! -  .2813594 !
!    .5590971 !
!   1.0143263 !
! -  .1810458 !
! - 1.1233045 !
```

Solution to a system of linear equations with a tri-diagonal matrix of coefficients

A tri-diagonal matrix is a square matrix such that the only non-zero elements are those in the main diagonal and those in the two diagonals immediately off the main one. A tri-diagonal matrix system will look as follows:

$$\begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 & 0 & 0 \\ 0 & a_{32} & a_{33} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n-2,n-2} & a_{n-2,n-1} & 0 \\ 0 & 0 & 0 & \cdots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & a_{n-1,n} & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-2} \\ b_{n-1} \\ b_n \end{bmatrix}$$

This system can also be written as **A**·**x** = **b**, where the nxn matrix **A**, and the nx1 vectors **x** and **b** are easily identified from the previous expression.

Since each column in the matrix of coefficients only uses three elements, we can enter the data as the elements of a nx3 matrix,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ \vdots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & a_{n-1,3} \\ 0 & a_{n,n-1} & a_{nn} \end{bmatrix}$$

Thomas algorithm for the solution of the tri-diagonal system of linear equations is an adaptation of the Gaussian elimination procedure. It consists of a forward elimination accomplished through the recurrence formulas:

$$a_{i2} = a_{i2} - a_{i1}\, a_{i-1,3}/a_{i-1,2},\ b_i = b_i - a_{i1}b_{i-1}/a_{i-1,2},\ \text{for i} = 2, 3, \ldots, n$$

The backward substitution step is performed through the following equations:

$$x_n = b_n/a_{n2},$$

and

$$x_i = (b_i - a_{i3}x_{i+1}/a_{i2}),\ \text{for i} = n\text{-}1, n\text{-}2, \ldots, 3, 2, 1\ \text{(counting backwards)}.$$

The following function implements the Thomas algorithm for the solution of a system of linear equations whose matrix of coefficients is a tri-diagonal matrix:

```
function [x] = Tridiag(A,b)
//Uses Thomas algorithm for tridiagonal matrices
//The matrix A is entered as a matrix of three columns and n rows
//containing the main diagonal and its two closest diagonals
[n m] = size(A);  // determine matrix size
//Check if number of columns = 3.
if m <> 3 then
       error('Matrix needs to have three columns only.')
       abort;
end
[nv mv] = size(b);  // determine vector size
if ((mv <> 1) | (nv <> n)) then
       error('Vector needs to be a column vector or it does not have same
number of rows as matrix')
end
//Recalculate matrix A and vector b
AA = A; bb = b;
for i = 2:n
       AA(i,2) = AA(i,2)-AA(i,1)*AA(i-1,3)/AA(i-1,2);
       bb(i)   = bb(i) - AA(i,1)*bb(i-1)/AA(i-1,2);
end;
//Back calculation of solution
x(n) = bb(n)/AA(n,2);
for i = n-1:-1:1
       x(i) = (bb(i)-AA(i,3)*x(i+1))/AA(i,2)
end;
//show solution
x
//end function
```


As an exercise, enter the following SCILAB commands:

```
-->A = [0,4,-1;-1,4,-1;-1,4,-1;-1,4,0]; b=[150;20;150;100];

-->getf('Tridiag')

-->x = Tridiag(A,b)
 x  =

!   44.976077 !
!   29.904306 !
!   54.641148 !
!   38.660287 !
```

Solution to tri-diagonal systems of linear equations using sparse matrices

Because a tri-diagonal matrix is basically a sparse matrix, we can use functions *lufact* and *lusolve* to obtain the solution to a tri-diagonal system of linear equations. The following function takes the nx3 matrix that represents a tri-diagonal matrix and produces the index and values of the corresponding sparse matrix:

```
function [index,val] = tritosparse(A)

//Converts the nx3 matrix A that represents
//a tri-diagonal matrix, into the corresponding
//nxn sparse matrix

krow = 0; kcol = 0; kval = 0;

krow = krow+1; row(krow) = 1;
kcol = kcol+1; col(kcol) = 1;
kval = kval+1; val(kval) = A(1,2)
krow = krow+1; row(krow) = 1;
kcol = kcol+1; col(kcol) = 2;
kval = kval+1; val(kval) = A(1,3)

for i = 2:n-1
       krow = krow+1; row(krow) = i;
       kcol = kcol+1; col(kcol) = i-1;
       kval = kval+1; val(kval) = A(i,1)
       krow = krow+1; row(krow) = i;
       kcol = kcol+1; col(kcol) = i;
       kval = kval+1; val(kval) = A(i,2)
       krow = krow+1; row(krow) = i;
       kcol = kcol+1; col(kcol) = i+1;
       kval = kval+1; val(kval) = A(i,3)
end;

krow = krow+1; row(krow) = n;
kcol = kcol+1; col(kcol) = n-1;
kval = kval+1; val(kval) = A(n,1)
krow = krow+1; row(krow) = n;
kcol = kcol+1; col(kcol) = n;
kval = kval+1; val(kval) = A(n,2)
```



```
index = [row col];

//end function
```

An application, using the compact tri-dimensional matrix A presented earlier is shown below:

First, we load the function:

```
-->getf('tritosparse')
```

A call to the function with argument A produces the values and index for a sparse matrix:

```
-->[index,val] = tritosparse(A)
 val  =
!   4. !
! - 1. !
! - 1. !
!   4. !
! - 1. !
! - 1. !
!   4. !
! - 1. !
! - 1. !
!   4. !

 index  =
!   1.    1. !
!   1.    2. !
!   2.    1. !
!   2.    2. !
!   2.    3. !
!   3.    2. !
!   3.    3. !
!   3.    4. !
!   4.    3. !
!   4.    4. !
```

Next, we put together the sparse matrix using the matrix *index* and the vector *val*:

```
-->As = sparse(index,val)
 As  =

(    4,    4) sparse matrix

(    1,    1)        4.
(    1,    2)      - 1.
(    2,    1)      - 1.
(    2,    2)        4.
(    2,    3)      - 1.
(    3,    2)      - 1.
(    3,    3)        4.
(    3,    4)      - 1.
(    4,    3)      - 1.
(    4,    4)        4.
```

To see the full form of the sparse matrix we use:

```
-->full(As)
```



```
 ans  =

!   4.  - 1.    0.    0. !
! - 1.    4.  - 1.    0. !
!   0.  - 1.    4.  - 1. !
!   0.    0.  - 1.    4. !
```

For the right-hand side vector previously defined we can use function *lusolve* to obtain the solution of the tri-diagonal system of linear equations:

```
-->x = lusolve(As,b)
 x  =

!   44.976077 !
!   29.904306 !
!   54.641148 !
!   38.660287 !
```

Iterative solutions to systems of linear equations

With a powerful numerical environment like SCILAB, which provides a number of pre-programmed functions to obtain the solution to systems of linear equations, there is really no need to use iterative methods. They are presented here only as programming exercises.

Iterative methods result from re-arranging the system of linear equations

$$
\begin{aligned}
a_{11}\cdot x_1 &+ a_{12}\cdot x_2 + a_{13}\cdot x_3 + \ldots + a_{1,n-1}\cdot x_{n-1} + a_{1,n}\cdot x_n = b_1,\\
a_{21}\cdot x_1 &+ a_{22}\cdot x_2 + a_{23}\cdot x_3 + \ldots + a_{2,n-1}\cdot x_{n-1} + a_{2,n}\cdot x_n = b_2,\\
&\quad\vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \ldots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \vdots\\
a_{n1}\cdot x_1 &+ a_{n2}\cdot x_2 + a_{n3}\cdot x_3 + \ldots + a_{n,n-1}\cdot x_{n-1} + a_{n,n}\cdot x_n = b_n.
\end{aligned}
$$

as

$$
\begin{aligned}
x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}x_n)/a_{11}\\
x_2 &= (b_2 - a_{22}x_2 - a_{23}x_3 - \ldots - a_{2n}x_n)/a_{22}\\
&\quad\vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\quad \ldots \qquad\quad \vdots \qquad\quad \vdots\\
x_n &= (b_n - a_{n2}x_2 - a_{n3}x_3 - \ldots - a_{nn}x_n)/a_{nn}
\end{aligned}
$$

The solution is obtained by first assuming a first guess for the solution $\mathbf{x}_{(0)} = [x_{10}, x_{20}, \ldots, x_{n0}]$, which is then replaced in the right-hand side of the recursive equations shown above to produce a new solution $\mathbf{x}_{(1)}$. With this new value in the right-hand side, a new approximation, $\mathbf{x}_{(2)}$ is obtained, and so on until the solution converges to a fixed value.

If you replace each value of x_i in the recursive equations as it gets calculated the approach is called the Gauss-Seidel method. An alternative, presented below as a SCILAB function, is the Jacobi iterative method.

Jacobi iterative method

In the Jacobi iterative method the new values of x_i's are replaced only once in each iteration. This method calculates the residual, **R**, from the equation **A**·**x** = **b**, as **R** = **A**·**x** - **b**, and adjusts the value of **R** recalculating **x** until the average of the residuals is smaller than a set value of tolerance, ε.

To program the Jacobi iterative method we create three files, as follows:

```
function [x]= Jacobi(A,b)
//Uses Jacobi iterative process to calculate x
getf('xbar.txt'); // this function calculates average absolute value
getf('Nextx.txt');// this function calculates a new value of x
[n m] = size(A);  // determine size of matrix A
// check if matrix is square
if n <> m then
       error('Matrix must be square');
       abort;
end;
//initialize variables
x = zeros(n,1);                          //x = [0 0 ... 0]
Itmax = 100.; eps = 0.001;               //max iterations and tolerance
Iter = 1; R = b-A*x; Rave = xbar(R);    //first value of residuals
disp(Iter, 'iteration'); disp(Rave,' Average residual:')
while ((Rave > eps) & (Iter < Itmax))   //Loop while no convergence
       x = Nextx(x,R,A)
       R = b-A*x;
       Rave = xbar(R);
       Iter = Iter + 1
       disp(Iter, 'iteration'); disp(Rave,' Average residual:')
end;
x
//end function Jacobi


function [xb] = xbar(x)
//Calculates average of absolute values of vector x
[n m] = size(x);
suma = 0.0;
for i = 1:n
       suma = suma + abs(x(i,1));
end;
xb = suma/n
//end function xbar


function [x]=Nextx(x,R,A)
//calculates residual average for the system A*x = b
[n m] = size(A);
for i = 1:n
       x(i,1) = x(i,1) + R(i,1)/A(i,i)
end;
```



```
x
//end function Nextx
```

As an exercise, within SCILAB enter the following:

```
-->A = [4 -1 0 0; -1 4 -1 0; 0 -1 4 -1; 0 0 -1 4]; b = [150; 200; 150; 100];
-->getf('Jacobi.txt')
-->Jacobi(A,b)
```

A few things to notice in these functions:

1. Function *Jacobi* loads the functions *xbar.txt* and *Nextx.txt*.
2. The function *size*, when used with a matrix or vector, requires two components. For that reason, whenever we use it in any of the functions *Jacobi*, *xbar* or *Nextx*, we always use `[n m] = size(…)`. Therefore, n = number of rows, m = number of columns.
3. To initialize the vector x we use the statement: `x = zeros(n,1)`. The general form of the function zeros is *zeros(m,n)*. See also, *help ones*.
4. The vector x having one column is treated as a matrix. See also the use of vector x in functions *xbar.txt* and *Nextx.txt*.
5. The function *disp* (display) prints the values in the list of variables, from the last to the first. Thus, the command `disp(iter, 'iteration: ')` prints a line with *'iteration:'* and the next line with the value of *iter*.
6. A while loop is used in function *Jacobi.txt*, which uses a compound logical expression, namely, `((Rave > eps) & (Iter < Itmax))`. So, as long as the average residual is larger than the tolerance (eps) and the number of iterations is less than the maximum number of iterations allowed (Itmax), the loop statements are repeated.
7. In function *xbar*, the word *suma* is used to hold a sum. The word *sum* is reserved by SCILAB (try: `help sum`) and cannot be redefined as a variable. By the way, *suma* is Spanish for sum.

REFERENCES (for all SCILAB documents at InfoClearinghouse.com)

Abramowitz, M. and I.A. Stegun (editors), 1965,"*Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*," Dover Publications, Inc., New York.

Arora, J.S., 1985, "*Introduction to Optimum Design*," Class notes, The University of Iowa, Iowa City, Iowa.

Asian Institute of Technology, 1969, "*Hydraulic Laboratory Manual*," AIT - Bangkok, Thailand.

Berge, P., Y. Pomeau, and C. Vidal, 1984,"*Order within chaos - Towards a deterministic approach to turbulence*," John Wiley & Sons, New York.

Bras, R.L. and I. Rodriguez-Iturbe, 1985,"*Random Functions and Hydrology*," Addison-Wesley Publishing Company, Reading, Massachussetts.

Brogan, W.L., 1974,"*Modern Control Theory*," QPI series, Quantum Publisher Incorporated, New York.

Browne, M., 1999, "*Schaum's Outline of Theory and Problems of Physics for Engineering and Science*," Schaum's outlines, McGraw-Hill, New York.

Farlow, Stanley J., 1982, "*Partial Differential Equations for Scientists and Engineers*," Dover Publications Inc., New York.

Friedman, B., 1956 (reissued 1990), "*Principles and Techniques of Applied Mathematics*," Dover Publications Inc., New York.

Gomez, C. (editor), 1999, "*Engineering and Scientific Computing with Scilab*," Birkhäuser, Boston.

Gullberg, J., 1997, "*Mathematics - From the Birth of Numbers*," W. W. Norton & Company, New York.

Harman, T.L., J. Dabney, and N. Richert, 2000, "*Advanced Engineering Mathematics with MATLAB® - Second edition*," Brooks/Cole - Thompson Learning, Australia.

Harris, J.W., and H. Stocker, 1998, "*Handbook of Mathematics and Computational Science*," Springer, New York.

Hsu, H.P., 1984, "*Applied Fourier Analysis*," Harcourt Brace Jovanovich College Outline Series, Harcourt Brace Jovanovich, Publishers, San Diego.

Journel, A.G., 1989, "*Fundamentals of Geostatistics in Five Lessons*," Short Course Presented at the 28th International Geological Congress, Washington, D.C., American Geophysical Union, Washington, D.C.

Julien, P.Y., 1998,"*Erosion and Sedimentation*," Cambridge University Press, Cambridge CB2 2RU, U.K.

Keener, J.P., 1988, "*Principles of Applied Mathematics - Transformation and Approximation*," Addison-Wesley Publishing Company, Redwood City, California.

Kitanidis, P.K., 1997,"*Introduction to Geostatistics - Applications in Hydogeology*," Cambridge University Press, Cambridge CB2 2RU, U.K.

Koch, G.S., Jr., and R. F. Link, 1971, "*Statistical Analysis of Geological Data - Volumes I and II*," Dover Publications, Inc., New York.

Korn, G.A. and T.M. Korn, 1968, "*Mathematical Handbook for Scientists and Engineers*," Dover Publications, Inc., New York.

Kottegoda, N. T., and R. Rosso, 1997, "*Probability, Statistics, and Reliability for Civil and Environmental Engineers*," The Mc-Graw Hill Companies, Inc., New York.

Kreysig, E., 1983, "*Advanced Engineering Mathematics - Fifth Edition*," John Wiley & Sons, New York.

Lindfield, G. and J. Penny, 2000, "*Numerical Methods Using Matlab®,"* Prentice Hall, Upper Saddle River, New Jersey.

Magrab, E.B., S. Azarm, B. Balachandran, J. Duncan, K. Herold, and G. Walsh, 2000, "*An Engineer's Guide to MATLAB®*", Prentice Hall, Upper Saddle River, N.J., U.S.A.

McCuen, R.H., 1989,"*Hydrologic Analysis and Design - second edition*," Prentice Hall, Upper Saddle River, New Jersey.

Middleton, G.V., 2000, "*Data Analysis in the Earth Sciences Using Matlab®,"* Prentice Hall, Upper Saddle River, New Jersey.

Montgomery, D.C., G.C. Runger, and N.F. Hubele, 1998, "*Engineering Statistics*," John Wiley & Sons, Inc.

Newland, D.E., 1993, "*An Introduction to Random Vibrations, Spectral & Wavelet Analysis - Third Edition*," Longman Scientific and Technical, New York.

Nicols, G., 1995, "*Introduction to Nonlinear Science*," Cambridge University Press, Cambridge CB2 2RU, U.K.

Parker, T.S. and L.O. Chua, , "*Practical Numerical Algorithms for Chaotic Systems*," 1989, Springer-Verlag, New York.

Peitgen, H-O. and D. Saupe (editors), 1988, "*The Science of Fractal Images*," Springer-Verlag, New York.

Peitgen, H-O., H. Jürgens, and D. Saupe, 1992, "*Chaos and Fractals - New Frontiers of Science*," Springer-Verlag, New York.

Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, 1989, "*Numerical Recipes - The Art of Scientific Computing (FORTRAN version)*," Cambridge University Press, Cambridge CB2 2RU, U.K.

Raghunath, H.M., 1985, "*Hydrology - Principles, Analysis and Design*," Wiley Eastern Limited, New Delhi, India.

Recktenwald, G., 2000, "*Numerical Methods with Matlab - Implementation and Application*," Prentice Hall, Upper Saddle River, N.J., U.S.A.

Rothenberg, R.I., 1991, "*Probability and Statistics*," Harcourt Brace Jovanovich College Outline Series, Harcourt Brace Jovanovich, Publishers, San Diego, CA.

Sagan, H., 1961,"*Boundary and Eigenvalue Problems in Mathematical Physics*," Dover Publications, Inc., New York.

Spanos, A., 1999,"*Probability Theory and Statistical Inference - Econometric Modeling with Observational Data*," Cambridge University Press, Cambridge CB2 2RU, U.K.

Spiegel, M. R., 1971 (second printing, 1999), "*Schaum's Outline of Theory and Problems of Advanced Mathematics for Engineers and Scientists*," Schaum's Outline Series, McGraw-Hill, New York.

Tanis, E.A., 1987, "*Statistics II - Estimation and Tests of Hypotheses*," Harcourt Brace Jovanovich College Outline Series, Harcourt Brace Jovanovich, Publishers, Fort Worth, TX.

Tinker, M. and R. Lambourne, 2000, "*Further Mathematics for the Physical Sciences*," John Wiley & Sons, LTD., Chichester, U.K.

Tolstov, G.P., 1962, "*Fourier Series*," (Translated from the Russian by R. A. Silverman), Dover Publications, New York.

Tveito, A. and R. Winther, 1998, "*Introduction to Partial Differential Equations - A Computational Approach*," Texts in Applied Mathematics 29, Springer, New York.

Urroz, G., 2000, "*Science and Engineering Mathematics with the HP 49 G - Volumes I & II*", www.greatunpublished.com, Charleston, S.C.

Urroz, G., 2001, "*Applied Engineering Mathematics with Maple*", www.greatunpublished.com, Charleston, S.C.

Winnick, J., , "*Chemical Engineering Thermodynamics - An Introduction to Thermodynamics for Undergraduate Engineering Students*," John Wiley & Sons, Inc., New York.

