2.3 Solution of network equations

Introduction

Sparse matrices are an important phenomenon in engineering. They occur regularly in network problems, and so, special methods used in their solution are of importance to us.

Let us consider a simple network with three nodes (that is, two node pairs) with each node connected to the other two. If we write the nodal equations, we will have

$$YV = I$$

Where Y is a 2 x 2 admittance matrix and V and I are 2 x 1 vectors. All elements of Y will be full (or have a non-zero entry.) However, if we take a circuit with ten nodes, with each node connected to three others, we will have a 9 x 9 admittance matrix with only a maximum of 36 non-zero elements, out of a total of 81. With a large network of (say) 1000 x 1000, it is possible to have less than 5000 non-zero elements, out of a total of one Million entries. This is one instance of how sparse matrices arise.

Common methods of solving matrix equations are quite inefficient in dealing with sparse matrices, and special methods are in use, which exploit their special features

We will first examine the most obvious solution of the equation we considered earlier:

$$YV=I$$
$$V = Y^{-1} I$$

where Y^{-1} is the inverse of Y. Matrix inversion is computationally very inefficient, even for a full matrix, for we have to evaluate the co-factor of each element of the matrix. This means that a (n-1) x (n-1) determinant has to be evaluated for each of the n^2 elements of the matrix, that is a total of (n-1)n! multiplications..

We will then look at Gaussian elimination as an algorithm for the solution of a matrix equation. We will also look at how equation ordering affects the accuracy of the solution.

Finally, we will look at LU factorisation and Cholesky factorisation

We have already examined the role of equation reordering and pivoting as a means of improving the accuracy of computation. When considering sparse matrices, we also need to be concerned about the need to conserve sparsity in the solution process, We have seen how inversion tends to almost completely fill up an originally sparse matrix, and that both Gaussian elimination and LU factorisation sometimes introduce new non-zero elements.

If we are interested in sparsity (as a means of reducing both storage requirements and computation time), we should consider special reordering schemes directed towards conserving sparsity. There are a variety of such schemes, each with its own merits and demerits. Some are very simple, and can be implemented with minimum time and effort, but are not very effective. They can be used when we are interested in only one run of the solution of a set of equations. More complex methods require relatively more effort, and can be justified when we have to resort to repeated runs.

A reordering scheme for sparse matrices to be useful will have to incorporate reordering techniques for both reduction of round off errors and for the preservation of sparsity.

Finally, we examine how a sparse matrix can be stored, so as to exploit its special features. In particular, we need to develop techniques of storage and retrieval that will reduce the total storage requirements while facilitating quick and easy data access –that is both writing and reading. These methods are together known as sparsity programming

2.3.1 Solution of linear state equations through Laplace transformation

Let us consider the system of state space equations:

$$\dot{x} = Ax + Bu$$

Laplace transformation of these yields:

$$sX(s) - x(0) = AX(s) + BU(s),$$

where

$$X(s) = \begin{bmatrix} X_1(s) \\ X_2(s) \\ . \\ . \\ X_n(s) \end{bmatrix}; \qquad x(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \\ . \\ . \\ x_n(0) \end{bmatrix}, \qquad U(s) = \begin{bmatrix} U_1(s) \\ U_2(s). \\ . \\ . \\ U_n(s) \end{bmatrix}$$

$$\therefore (sI - A)X(s) = x(0) + BU(s)$$
$$ie.,\ X(s) = (sI - A)^{-1}x(0) + (sI - A)^{-1}BU(s)$$
$$\therefore x(t) = L^{-1}\{(sI - A)^{-1}x(0)\} + L^{-1}\{(sI - A)^{-1}BU(s)\}$$

The solution consists of two parts:

- $L^{-1}\{(sI - A)^{-1}x(0)\}$, which is the contribution made by the initial conditions. This is a transient, but it is not the complete transient.
- $L^{-1}\{(sI - A)^{-1}BU(s)\}$, which is the contribution made by the inputs to the system.

This contribution consists of two parts itself, a transient term and a steady state term. Both these contain the term $(sI - A)^{-1}$.
Now,

$$(sI = A)^{-1} = \frac{adj\ (sI - A)}{|sI - A|}$$

$|sI - A|$ is a polynomial in s, of degree n.
Each element in $adj\ (sI - A)$ is a polynomial in s, of degree (n-1) or less.

Each element of $\frac{adj\ (sI - A)}{|sI - A|}$ can be split up into partial fractions of the form:

$$\frac{b_1}{s - \lambda_1} + \frac{b_2}{s - \lambda_2} + \ . \ . \ . \ + \frac{b_n}{s - \lambda_n}$$

if the roots of $|sI - A| = 0$ are distinct, or

$$\frac{b_1}{s - \lambda_1} + \ . \ . \ . \ + \frac{b_p}{s - \lambda_p} + \frac{b_{p+1}}{(s - \lambda_p)^2} + \ . \ . \ . \ + \frac{b_{p+q-1}}{(s - \lambda_p)^q} + \ . \ . \ . \ . \ + \frac{b_n}{s - \lambda_{n-q+1}}$$

if there is one root of multiplicity q.

These λ's are the eigen-values of the system, and $|sI - A| = 0$ is the characteristic equation of the system. The eigen-values are the roots of the characteristic equation. $|sI - A|$ is known as the characteristic polynomial. The eigen-values may be either real, or they occur in complex conjugate pairs.

Example:

$$A = \begin{bmatrix} 1 & 0 & -1 \\ 1 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix},$$

$$x(0) = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}; \quad U = \begin{bmatrix} H(t) \\ H(t) \end{bmatrix}$$

$$sI - A = \begin{bmatrix} s-1 & 0 & 0 \\ -1 & s+1 & 0 \\ 0 & 0 & s+1 \end{bmatrix}$$

$$|sI - A| = (s-1)(s+1)^2$$

$$[sI - A]^{-1} = \frac{adj\,(sI - A)}{|sI - A|}$$

$$= \begin{bmatrix} \frac{1}{s-1} - \frac{1}{s^2-1} \\ \frac{1}{s^2-1} - \frac{1}{(s-1)(s+1)^2} \\ \frac{1}{s+1} \end{bmatrix} + \begin{bmatrix} \frac{1}{s(s-1)} - \frac{1}{s(s^2-1)} \\ \frac{1}{s(s^2-1)} - \frac{1}{s(s-1)(s+1)^2} \\ \frac{1}{s(s+1)} \end{bmatrix}$$

$$= \frac{1}{(s-1)(s+1)^2} \begin{bmatrix} (s+1)^2 & 0 & -(s+1) \\ s+1 & s^2-1 & -1 \\ 0 & 0 & s^2-1 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{s-1} & 0 & \frac{-1}{s^2-1} \\ \frac{1}{s^2-1} & \frac{1}{s+1} & \frac{-1}{(s-1)(s+1)^2} \\ 0 & 0 & \frac{1}{s+1} \end{bmatrix}$$

The Laplace transform of the response due to the initial conditions =
$(sI - A)^{-1}x(0)$

$$= \begin{bmatrix} \frac{1}{s-1} & 0 & \frac{-1}{s^2-1} \\ \frac{1}{s^2-1} & \frac{1}{s+1} & \frac{-1}{(s-1)(s+1)^2} \\ 0 & 0 & \frac{1}{s+1} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{s-1} - \frac{1}{s^2-1} \\ \frac{1}{s^2-1} - \frac{1}{(s-1)(s+1)^2} \\ \frac{1}{s+1} \end{bmatrix}$$

The Laplace transform of the response due to the input =
$(sI - A)^{-1}BU(s)$

$$= \begin{bmatrix} \frac{1}{s-1} & 0 & \frac{-1}{s^2-1} \\ \frac{1}{s^2-1} & \frac{1}{s+1} & \frac{-1}{(s-1)(s+1)^2} \\ 0 & 0 & \frac{1}{s+1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{s} \\ \frac{1}{s} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{s-1} & 0 & \frac{-1}{s^2-1} \\ \frac{1}{s^2-1} & \frac{1}{s+1} & \frac{-1}{(s-1)(s+1)^2} \\ 0 & 0 & \frac{1}{s+1} \end{bmatrix} \begin{bmatrix} \frac{1}{s} \\ 0 \\ \frac{1}{s} \end{bmatrix} = \begin{bmatrix} \frac{1}{s(s-1)} - \frac{1}{s(s^2-1)} \\ \frac{1}{s(s^2-1)} - \frac{1}{s(s-1)(s+1)^2} \\ \frac{1}{s(s+1)} \end{bmatrix}$$

Then, the transform of the total response
= The transform of the response due to initial conditions
+ The transform of the response due to the input.

$$= \begin{bmatrix} \dfrac{1}{s-1} + \dfrac{1}{s+1} \\ \dfrac{1}{2(s-1)} - \dfrac{1}{2(s+1)} \\ \dfrac{1}{s} \end{bmatrix}$$

In the time domain, the response =

$$L^{-1} \begin{bmatrix} \dfrac{1}{s-1} + \dfrac{1}{s+1} \\ \dfrac{1}{2(s-1)} - \dfrac{1}{2(s+1)} \\ \dfrac{1}{s} \end{bmatrix} = \begin{bmatrix} e^{t} + e^{-t} \\ \dfrac{1}{2}(e^{t} - e^{-t}) \\ 1 \end{bmatrix} H(t)$$

2.3.2 Solution of transient equations

We have studied about the methods for the solution of linear algebraic equations that arise in the steady state solution of networks. We will now look at how differential equations describing the transient behaviour of networks may be handled.

Earlier, we studied about the dynamic representation of networks, through the formulation of state space equations. Our treatment of electrical circuits was limited to time-invariant systems, in that we assumed that parameters such as the resistance, inductance or capacitance of an element were not functions of time. We will continue with this assumption, and restrict our treatment to time-invariant systems.

Analytical methods for the solution of systems of differential equations exist only for a limited class of simple, linear equations. For the study of more complex and non-linear systems, we need to convert the differential equations to difference equations, and then apply numerical techniques for their solution. We will study analytical methods for the solution of systems of differential equations and also some numerical techniques for the solution of systems of difference equations.

Analytical solution of linear state equations

We have already noted the relationship between the state space representation and the s-plane representation of a system. One approach to the solution of state equations is through its Laplace transform.

Another approach would be through the evaluation of the matrix exponential.

As is to be expected, both these solutions are strongly influenced by the eigen-values of the system.

Numerical solution of state equations: Solution of linear state equations through the matrix exponential

We have seen that the solution to
$\dot{x} = Ax + Bu$ is

$$x(t) = L^{-1}\left\{(sI - A)^{-1}x(0)\right\} + L^{-1}\left\{(sI - A)^{-1}BU(s)\right\}$$

Let $L^{-1}\left\{(sI - A)^{-1}\right\} = \Phi(t)$

Therefore, (by the convolution theorem):

$$x(t) = \Phi(t)\,x(0) + \int_0^t \Phi(t-\tau)\,Bu(\tau)\,d\tau$$

We now need to evaluate Φ(t).

We will assume a power series solution for the homogeneous equation $\dot{x} = Ax$, of the form:

$$x(t) = a_0 + a_1 t + a_2 t^2 + . \ . \ . \ .$$

This gives:

$$\begin{aligned}\dot{x}(t) \quad &= a_1 + 2a_2 t + 3a_3 t^2 + . \ . \ . \\ &= Ax = A(a_0 + a_1 t + a_2 t^2 + . \ . \ .)\end{aligned}$$

Equating coefficients of powers of t, we have:

$$
\begin{aligned}
a_1 &= Aa_0 \\
a_2 &= \frac{1}{2}Aa_1 = \frac{1}{2}A\,Aa_0 = \frac{1}{2}A^2a_0 \\
a_3 &= \frac{1}{3}Aa_2 = \frac{1}{3}A\,\frac{1}{2}A^2a_0 = \frac{1}{2.3}A^3a_0 \\
&. \\
&. \\
a_r &= \frac{1}{r!}A^r a_0
\end{aligned}
$$

We also have, by substitution t=0 in our power series,

$a_0 = x(0)$

This gives us the solution:

$$
\begin{aligned}
x(t) &= [I + At + \frac{1}{2!}A^2t^2 + \frac{1}{3!}A^3t^3 + .\ .\ .\ .]x(0) \\
&= e^{At}\,x(0)
\end{aligned}
$$

The solution of the complete equation
$\dot{x} = Ax + Bu$ is:

$$
\begin{aligned}
x(t) &= \Phi(t)\,x(0) + \int_0^t \Phi(t-\tau)Bu(\tau)d\tau \\
&= e^{At}\,x(0) + \int_0^t e^{A(t-\tau)}Bu(\tau)d\tau \\
&= e^{At}x(0) + e^{At}\int_0^t e^{-A\tau}\,Bu(\tau)d\tau
\end{aligned}
$$

Matrix inversion

We will consider the following equation, which we have already encountered:

$$
\begin{bmatrix} G_1 + G_2 + G_6 & -G_2 & -G_6 \\ -G_2 & G_2 + G_3 + G_4 & -G_4 \\ -G_6 & -G_4 & G_4 + G_5 + G_6 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} i_s \\ 0 \\ 0 \end{bmatrix}
$$

Let us assume some numerical values for each G_i and for i_s.
$G_1 = g_3 = G_5 = 1$,
$G_2 = G_4 = G_6 = 2$,
I_s=1

Then the equations would be:

$$\begin{bmatrix} 5 & -2 & -2 \\ -2 & 5 & -2 \\ -2 & -2 & 5 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

To compute the inverse of this matrix, we need to first compute its determinant:

Δ = 5(25-4) + 2(-10-4)-2(4+10) = 49

We then have to compute the co-factor of each element Δ_{ij} to obtain:

$$G^{-1} = \frac{1}{49}\begin{bmatrix} (25-4) & -(-10-4) & (4+10) \\ -(-10-4) & (25-4) & -(-10-4) \\ (4+10) & -(-10-4) & (25-4) \end{bmatrix}$$
$$= \frac{1}{49}\begin{bmatrix} 21 & 14 & 14 \\ 14 & 21 & 14 \\ 14 & 14 & 21 \end{bmatrix}$$
$$= \begin{bmatrix} 0.4286 & 0.2857 & 0.2857 \\ 0.2857 & 0.4286 & 0.2857 \\ 0.2857 & 0.2857 & 0.4286 \end{bmatrix}$$

Now, writing

$V=G^{-1} I$,

We have:

$$\begin{bmatrix} v_1 \\ v_2 \\ v \end{bmatrix} = \begin{bmatrix} 0.4286 \\ 0.2857 \\ 0.2857 \end{bmatrix}$$

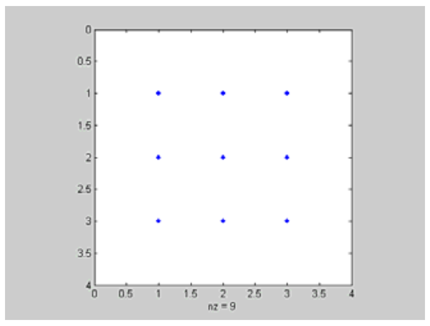
We can use MATLAB to obtain this result using:

```
G=[5 -2 -2;-2 5 -2;-2 -2 5]

G =

   5  -2  -2
  -2   5  -2
  -2  -2   5
```

```
is=[1;0;0]

is =

     1
     0
     0

v=inv(G)*is

v =

    0.4286
    0.2857
    0.2857
```

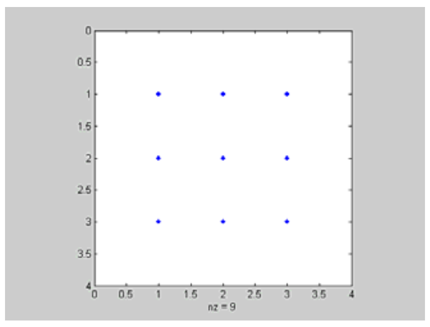
We can use the “spy” instruction to plot the non-zero elements of G:

```
Spy(G)
```



and of G^{-1}:

```
Spy(inv(G))
```

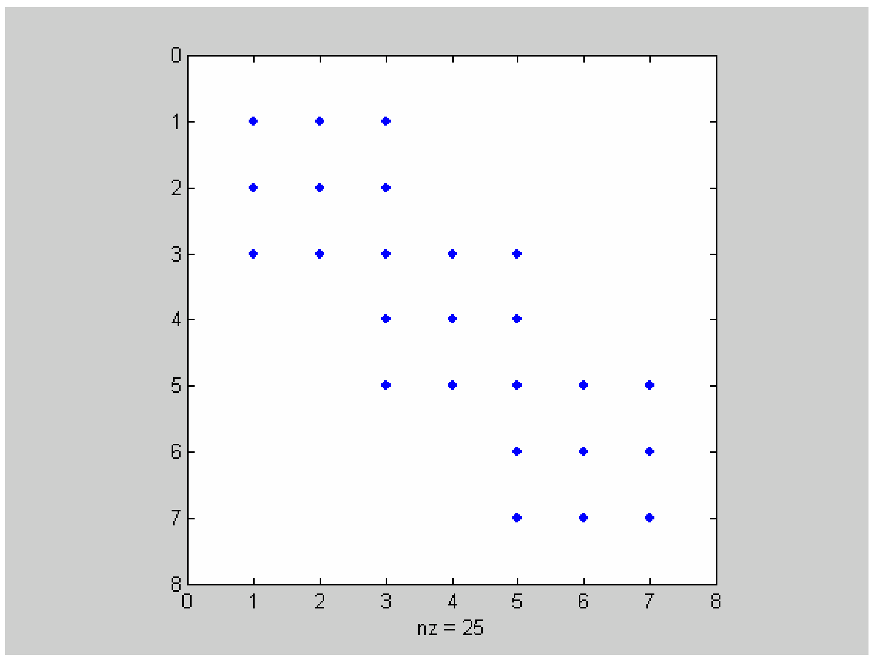


They are both full matrices and nothing (in terms of storage etc.) is gained or lost.

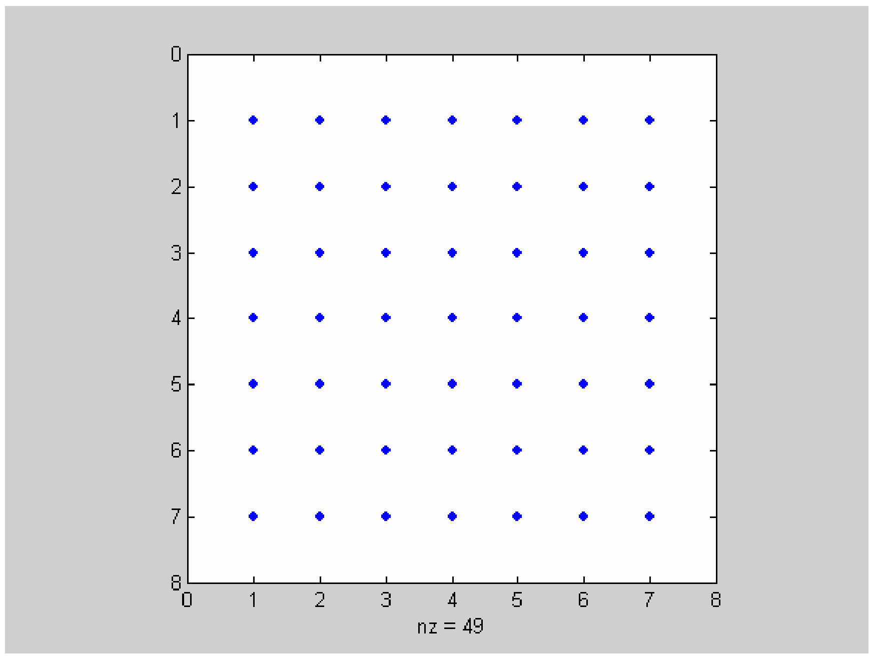
Now lets look at the next example we considered, of three such networks connected in cascade. The non-zero elements of the original matrix and of its inverse are as shown:

$$G = \begin{bmatrix} 5 & -2 & -2 & & & & \\ -2 & 5 & -2 & & & & \\ -2 & -2 & 10 & -2 & -2 & & \\ & & -2 & 5 & -2 & & \\ & & -2 & -2 & 10 & -2 & -2 \\ & & & & -2 & 5 & -2 \\ & & & & -2 & -2 & 5 \end{bmatrix}$$

```
G=[5 -2 -2 0 0 0 0;-2 5 -2 0 0 0 0;
-2 -2 10 -2 -2 0 0;0 0 -2 5 -2 0 0;
0 0 -2 -2 10 -2 -2;0 0 0 0 -2 5 -2;
0 0 0 0 -2 -2 5]

G =

    5   -2   -2    0    0    0    0
   -2    5   -2    0    0    0    0
   -2   -2   10   -2   -2    0    0
    0    0   -2    5   -2    0    0
    0    0   -2   -2   10   -2   -2
    0    0    0    0   -2    5   -2
    0    0    0    0   -2   -2    5

>> spy(G)
```



```
>> spy(inv(G))
```



```
>> inv(G)

ans =

  Columns 1 through 5

    0.3214    0.1786    0.1250    0.0714    0.0536
    0.1786    0.3214    0.1250    0.0714    0.0536
    0.1250    0.1250    0.1875    0.1071    0.0804
    0.0714    0.0714    0.1071    0.2857    0.1071
    0.0536    0.0536    0.0804    0.1071    0.1875
    0.0357    0.0357    0.0536    0.0714    0.1250
    0.0357    0.0357    0.0536    0.0714    0.1250

  Columns 6 through 7

    0.0357    0.0357
    0.0357    0.0357
    0.0536    0.0536
    0.0714    0.0714
    0.1250    0.1250
    0.3214    0.1786
    0.1786     0.3214
    0.1787
>> nnz(G)

ans =

    25

>> nnz(inv(G))

ans =

    49
```

We see that the original matrix had only 25 non-zero elements while the inverse has 49 non-zero elements, and is full.

Gaussian elimination

We will study this algorithm through the example we have been considering:

$$\begin{bmatrix} 5 & -2 & -2 \\ -2 & 5 & -2 \\ -2 & -2 & 5 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Step 1: Divide the first row by its diagonal element:

$$\begin{bmatrix} 1 & -2/5 & -2/5 \\ -2 & 5 & -2 \\ -2 & -2 & 5 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1/5 \\ 0 \\ 0 \end{bmatrix}$$

Eliminate v_1 from the other equations by subtracting the relevant multiples of equation 1 from the others:

$$\begin{bmatrix} 1 & -2/5 & -2/5 \\ 0 & 5-4/5 & -2-4/5 \\ 0 & -2-4/5 & 5-4/5 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1/5 \\ 2/5 \\ 2/5 \end{bmatrix}$$

We now repeat the process with the second row, that is first, make the diagonal element unity, then eliminate the second variable from the third equation:

$$\begin{bmatrix} 1 & -2/5 & -2/5 \\ 0 & 1 & -2/3 \\ 0 & -14/5 & 21/5 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1/5 \\ 2/21 \\ 2/5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -2/5 & -2/5 \\ 0 & 1 & -2/3 \\ 0 & 0 & 21/5-(14/5)(2/3) \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1/5 \\ 2/21 \\ 2/5+(14/5)(2/21) \end{bmatrix}$$

Simplifying:

$$\begin{bmatrix} 1 & -2/5 & -2/5 \\ 0 & 1 & -2/3 \\ 0 & 0 & 7/3 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1/5 \\ 2/21 \\ 2/3 \end{bmatrix}$$

Now, normalising the last equation, we have:

$$\begin{bmatrix} 1 & -2/5 & -2/5 \\ 0 & 1 & -2/3 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1/5 \\ 2/21 \\ 2/7 \end{bmatrix}$$

This gives the results as:

$V_3 = 2/7$
$V_2 = 2/21 + 2/3\ V_3 = 2/21 + 4/21 = 2/7$
$V_1 = 1/5 + 2/5\ V_2 + 2/5\ V_3 = 1/5 + 8/35 = 3/7$

We are now in a position to attempt to write down the general algorithm. Consider the (n x n) matrix A and (n x 1) vectors x and b, where x is the unknown.

$A_{nxn}x_{nx1} = b_{nx1}$

Our strategy is to eliminate x_1 from all the (n-1) equations, other than the first. To do this, we first divide the first equation throughout by a_{11}, so that the revised a_{11} is equal to 1.

```
For i = 1 to n:
        $a_{1i} = a_{1i} / a_{11}$
        $b_1 = b_1 / a_{11}$
```

Then, for each of the rows 2 to n, we subtract a_{i1} times the first row from each term, in other words:

```
For i = 2 to n:
        $b_i = b_i - a_{i1} \times b_1$
        For j = 1 to n:
                $a_{ij} = a_{ij} - a_{i1} \times a_{1j}$
```

This would mean that x_1 is eliminated from all but the first equation, so that we are left with (n-1) equations in (n-1) unknowns. We can then repeat the same algorithm for the new (n – 1) x ((n – 1) matrix. Finally, we will be left with only one equation, corresponding to the last variable x_n:

$x_n = b_n$

The rest of the algorithm consists of the back-substitution process, whereby x_{n-1} is calculated using the known value of x_n, and then x_{n-2} is calculated, and so on until we obtain all values up to x_1.

$x_{n-1} = b_{n-1} - a_{n-1,n}\ x_n$

For the general case:

$$x_i = b_i - \sum_{j=i+1}^{n} a_{ij} x_j, \qquad i = n-1, n-2, \ldots, 1$$

This algorithm suffers from the disadvantage that the solution has to be repeated from the very beginning even when the matrix A has not changed at all, but only the vector b has changed. We can overcome this difficulty by actually not carrying out the operations on b during the forward reduction, but keeping a record of the necessary operations. This philosophy has lead to the development of algorithms such as the LU factorisation.

The other main problem is that of ill-conditioned or badly ordered matrices.

Re-ordering the equations (row pivoting) or the variables (column pivoting) can help to resolve problems with bad ordering.

LU Factorisation

We will consider the same example as before:

$$\begin{bmatrix} 5 & -2 & -2 \\ -2 & 5 & -2 \\ -2 & -2 & 5 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

We would wish to be able to avoid some of the disadvantages of Gaussian elimination, in particular, the necessity to re-do all the computations in case of having to estimate [v] for a different [i], A remaining the same.

Let us assume that we van find two matrices L and U such that:

L*U = A

L and U being lower triangular and upper triangular, respectively. Then, it would be easy to compute x satisfying:

L*U*x = b

in two steps. First we find y such that:

L*y = b

Then, x such that:

U*x = y

For the example chosen:

$$\begin{bmatrix} 5 & -2 & -2 \\ -2 & 5 & -2 \\ -2 & -2 & 5 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix}$$

We can now write down each of these terms, almost by inspection:

$$u_{11} = 5$$
$$u_{12} = -2$$
$$u_{13} = -2$$
$$l_{21}u_{11} = -2 \Rightarrow l_{21} = -2/5$$
$$l_{21}u_{12} + u_{22} = 5 \Rightarrow u_{22} = 5 - 4/5 = 21/5$$
$$l_{21}u_{13} + u_{23} = -2 \Rightarrow u_{23} = -2 - 4/5 = -14/5$$

$$l_{31}u_{11} = -2 \Rightarrow l_{31} = -2/5$$
$$l_{31}u_{12} + l_{32}u_{22} = -2 \Rightarrow l_{32} = -2/3$$
$$l_{31}u_{13} + l_{32}u_{23} + u_{33} = 5 \Rightarrow u_{33} = 7/3$$

We have now completed the computation of the factored form:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -2/5 & 1 & 0 \\ -2/5 & -2/3 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 5 & -2 & -2 \\ 0 & 21/5 & -14/5 \\ 0 & 0 & 7/3 \end{bmatrix}$$

We can check whether the factorisation is correct by multiplying L by U using MATLAB:

```
L=[1 0 0;-2/5 1 0;-2/5 -2/3 1];
U=[5 -2 -2;0 21/5 -14/5;0 0 7/3];
A=L*U
A =

   5.0000  -2.0000  -2.0000
  -2.0000   5.0000  -2.0000
  -2.0000  -2.0000   5.0000
```

We can also use MATLAB to perform the LU factorisation:

```
[L,U]=lu(A)

L =

   1.0000             0                 0
  -0.4000             1.0000            0
  -0.4000            -0.6667     1.0000


U =

   5.0000            -2.0000           -2.0000
        0             4.2000           -2.8000
        0             0                 2.3333
```

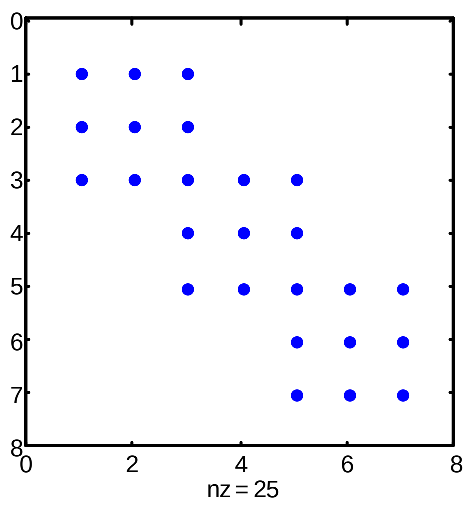
When we need to optimise the use of storage, it is possible to store all the values in one matrix, as it is not necessary to store either the zeros or the 1s. There is also a distinct MATLAM command for this:

```
lu(A)

ans =

   5.0000  -2.0000  -2.0000
   0.4000   4.2000  -2.8000
   0.4000   0.6667   2.3333
```

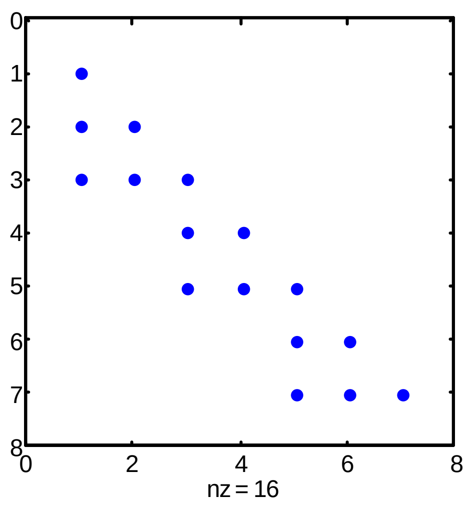
Let us now see the effect of LU factorisation on a sparse matrix.
We will again use the example we considered earlier.

```
A =

    5   -2   -2    0    0    0    0
   -2    5   -2    0    0    0    0
   -2   -2   10   -2   -2    0    0
    0    0   -2    5   -2    0    0
    0    0   -2   -2   10   -2   -2
    0    0    0    0   -2    5   -2
    0    0    0    0   -2   -2    5
```

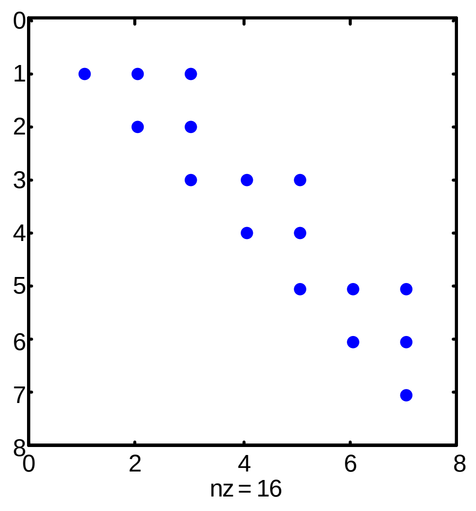
```
spy(A)
```



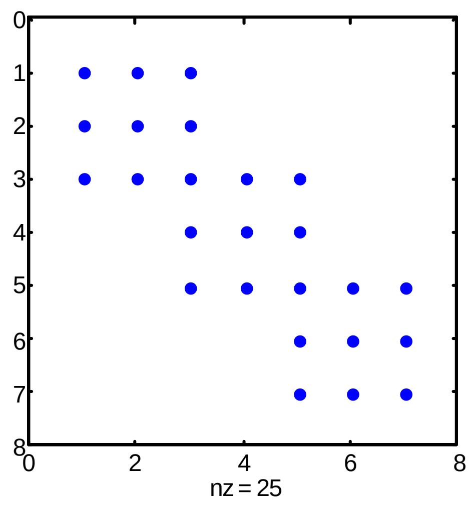
```
» [L,U]=lu(A)

L =

    1.0000         0         0         0         0         0         0
   -0.4000    1.0000         0         0         0         0         0
   -0.4000   -0.6667    1.0000         0         0         0         0
         0         0   -0.2727    1.0000         0         0         0
         0         0   -0.2727   -0.5714    1.0000         0         0
         0         0         0         0   -0.2500    1.0000         0
         0         0         0         0   -0.2500   -0.5556    1.0000

U =

    5.0000   -2.0000   -2.0000         0         0         0         0
         0    4.2000   -2.8000         0         0         0         0
         0         0    7.3333   -2.0000   -2.0000         0         0
         0         0         0    4.4545   -2.5455         0         0
         0         0         0         0    8.0000   -2.0000   -2.0000
         0         0         0         0         0    4.5000   -2.5000
         0         0         0         0         0         0    3.1111

spy(L)
```



```
spy(U)
```



spy(lu(A))

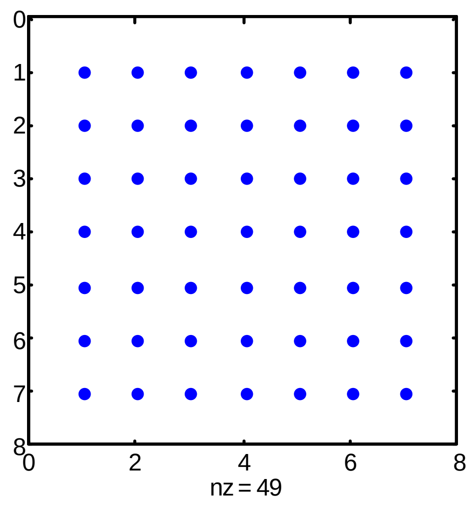


Notice that in this particular case, there has been no increase in storage requirements. This is only if we use one matrix to store both lower and upper triangles, with implied storage of zero and unity values.

This is not always the case, and some non-zero elements may be introduced during factorisation.

Compare this with the result we obtained with inversion, where the reslting matrix was a full matrix.

spy(inv(A))



Cholesky factorisation

Unlike the LU factorisation, this works only for symmetric positive definite matrices. Similar to the procedure we adopted for the computation of the LU factorisation, we can start with the expected result to obtain the factorisation algorithm.

The Cholesky factorisation of a symmetric positive definite matrix A produces two factors such that:

A = C’ * C

We will use the MATLAB command to obtain the factors of the matrix we considered earlier:

```
» A=[5 -2 -2 0 -1 0 0;
-2 5 -2 0 0 0 0;
-2 -2 11 -2 -2 0 -1;
0 0 -2 5 -2 0 0;
-1 0 -2 -2 11 -2 -2;
0 0 0 0 -2 5 -2;
0 0 -1 0 -2 -2 5]

A =

     5    -2    -2     0    -1     0     0
    -2     5    -2     0     0     0     0
    -2    -2    11    -2    -2     0    -1
     0     0    -2     5    -2     0     0
    -1     0    -2    -2    11    -2    -2
     0     0     0     0    -2     5    -2
     0     0    -1     0    -2    -2     5

» C=chol(A)

C =

    2.2361   -0.8944   -0.8944         0   -0.4472         0         0
         0    2.0494   -1.3663         0   -0.1952         0         0
         0         0    2.8868   -0.6928   -0.9238         0   -0.3464
         0         0         0    2.1260   -1.2418         0   -0.1129
         0         0         0         0    2.8925   -0.6914   -0.8505
         0         0         0         0         0    2.1265   -1.2171
         0         0         0         0         0         0    1.6317

» transpose(C)*C

ans =

    5.0000   -2.0000   -2.0000         0   -1.0000         0         0
   -2.0000    5.0000   -2.0000         0         0         0         0
   -2.0000   -2.0000   11.0000   -2.0000   -2.0000         0   -1.0000
         0         0   -2.0000    5.0000   -2.0000         0         0
   -1.0000         0   -2.0000   -2.0000   11.0000   -2.0000   -2.0000
         0         0         0         0   -2.0000    5.0000   -2.0000
         0         0   -1.0000         0   -2.0000   -2.0000    5.0000
```

We will use the MATLAB command nnz to obtain the number of non-zero elements of A and C:

```
» nnz(A)

ans =

   29

» nnz(C)

ans =

   20
```

We will now examine the effect of re-ordering the equations on sparsity. We will use the reordering algorithm symrcm available in MATLAB. Its description is as follows:

```
SYMRCM Symmetric reverse Cuthill-McKee permutation.
   p = SYMRCM(S) returns a permutation vector p such that S(p,p)
   tends to have its diagonal elements closer to the diagonal than S.
   This is a good preordering for LU or Cholesky factorization of
   matrices that come from "long, skinny" problems.  It works for
   both symmetric and asymmetric S.
```

```
» p=symrcm(A)

p =

    2    1    7    6    3    5    4

» A1=A(p,p)

A1 =

    5   -2    0    0   -2    0    0
   -2    5    0    0   -2   -1    0
    0    0    5   -2   -1   -2    0
    0    0   -2    5    0   -2    0
   -2   -2   -1    0   11   -2   -2
    0   -1   -2   -2   -2   11   -2
    0    0    0    0   -2   -2    5

» chol(A1)

ans =

   2.2361   -0.8944         0         0   -0.8944         0         0
        0    2.0494         0         0   -1.3663   -0.4880         0
        0         0    2.2361   -0.8944   -0.4472   -0.8944         0
        0         0         0    2.0494   -0.1952   -1.3663         0
        0         0         0         0    2.8452   -1.1716   -0.7029
        0         0         0         0         0    2.5928   -1.0890
        0         0         0         0         0         0    1.8221
```

```
» C1=chol(A1)

C1 =

   2.2361   -0.8944        0        0   -0.8944        0        0
        0    2.0494        0        0   -1.3663   -0.4880        0
        0         0   2.2361   -0.8944   -0.4472   -0.8944        0
        0         0        0    2.0494   -0.1952   -1.3663        0
        0         0        0         0    2.8452   -1.1716   -0.7029
        0         0        0         0         0    2.5928   -1.0890
        0         0        0         0         0         0    1.8221

» nnz(C1)

ans =

   19
```

Compare with a different reordering algorithm:

SYMMMD Symmetric minimum degree permutation.
   p = SYMMMD(S), for a symmetric positive definite matrix S,
   returns the permutation vector p such that S(p,p) tends to have a
   sparser Cholesky factor than S. Sometimes SYMMMD works well
   for symmetric indefinite matrices too.

```
» q=symmmd(A)

q =

    4    1    2    6    7    3    5

» A2=A(q,q)

A2 =

    5    0    0    0    0   -2   -2
    0    5   -2    0    0   -2   -1
    0   -2    5    0    0   -2    0
    0    0    0    5   -2    0   -2
    0    0    0   -2    5   -1   -2
   -2   -2   -2    0   -1   11   -2
   -2   -1    0   -2   -2   -2   11

» C2=chol(A2)

C2 =

   2.2361        0        0        0        0   -0.8944   -0.8944
        0   2.2361   -0.8944        0        0   -0.8944   -0.4472
        0        0    2.0494        0        0   -1.3663   -0.1952
        0        0        0    2.2361   -0.8944        0   -0.8944
        0        0        0        0    2.0494   -0.4880   -1.3663
        0        0        0        0        0    2.7010   -1.5303
        0        0        0        0        0        0    2.2256
```

```
» nnz(C2)

ans =

   19
```

We end up with the same number of non-zero elements after factorisation. We will now try a very simple reordering algorithm: reorder by rank of non-zero elements in each row.

```
» r=[2 4 6 1 7 3 5]

r =

    2    4    6    1    7    3    5

» A3=A(r,r)

A3 =

    5    0    0   -2    0   -2    0
    0    5    0    0    0   -2   -2
    0    0    5    0   -2    0   -2
   -2    0    0    5    0   -2   -1
    0    0   -2    0    5   -1   -2
   -2   -2    0   -2   -1   11   -2
    0   -2   -2   -1   -2   -2   11

» C3=chol(A3)

C3 =

   2.2361        0        0  -0.8944        0  -0.8944        0
        0   2.2361        0        0        0  -0.8944  -0.8944
        0        0   2.2361        0  -0.8944        0  -0.8944
        0        0        0   2.0494        0  -1.3663  -0.4880
        0        0        0        0   2.0494  -0.4880  -1.3663
        0        0        0        0        0   2.7010  -1.5303
        0        0        0        0        0        0   2.2256

» nnz(C3)

ans =

   18
```

The reordering in terms of the rank order of non-zero elements gives the best result.

Solution of ordinary differential equations

Circuits containing energy storage elements (capacitors and inductors) give rise to systems of equations containing derivatives of currents and / or voltages. The numerical solution of such equations is based on their conversion to difference equations, using approximate representations.

Runga-Kutta methods

There is a family of Runga-Kutta methods, each based on the Taylor series, but differing by the number of terms of the series considered. The simplest of these is the second-order Runga-Kutta method, which takes on one more term than the Euler method:

$$x(t_0+h) = x(t_0) + h\,\dot{x}(t_0) + \frac{h^2}{2!}\ddot{x}(t_0) + R_3$$

We now use the first order approximation to compute the second derivative as:

$$\ddot{x}(t_0) \quad \approx \frac{\dot{x}(t_0+h) - \dot{x}(t_0)}{h}$$

Substituting this in the first equation, we get:

$$x(t_0+h) \approx x(t_0) + h\,\dot{x}(t_0) + \frac{h^2}{2}\frac{\dot{x}(t_0+h) - \dot{x}(t_0)}{h} = x(t_0) + h\;\frac{\dot{x}(t_0) + \dot{x}(t_0+h)}{2}$$

The resulting algorithm for the second order Runga-Kutta method is therefore as follows:

Start with the initial value, $x(t_0)$.

Evaluate $\dot{x}(t_0) = Ax(t_0) + Bu(t_0)$

Compute $\hat{x}_1(t_0+h) = x(t_0) + h\,\dot{x}(t_0)$

Evaluate $\dot{\hat{x}}_1(t_0+h) = A\hat{x}_1(t_0+h) + Bu(t_0+h)$

Compute $\hat{x}_2(t_0+h) = x(t_0) + \frac{h}{2}[\dot{x}(t_0) + \dot{\hat{x}}_1(t_0+h)]$

Set t_0 = (t_0+h) and go back to step 1.

The most popular algorithm is the fourth order Runga-Kutta method, which uses two more terms of the Taylor series expansion to obtain a more accurate estimation.

You would have noted that the Taylor series also gives an estimated upper bound of the error. This is used to implement a dynamic step length adjustment algorithm, whereby the step length is halved if the estimate of the error exceeds a design value, and is doubled if it falls below a specified lower limit. The doubling of the step length is used to reduce computation effort and to reduce numerical round-off errors. In practice, we need to trade-off between the two types of errors to get a best estimate.

One major advantage of numerical methods is that it is not limited to linear systems. Even though we implicitly assumed the system to be linear, by considering the system equation to be

$$\dot{x} = Ax + Bu\,;$$

this is not necessary. We could evaluate the derivative of x using any linear or non-linear expression:

$$\dot{x} = f(x,u)\,.$$

and the algorithm would still work.

The formula for the fourth order Runga-Kutta method is as follows:

$$x(t+\tau) = x(t) + \frac{h}{6}(f(t) + 4f(t+\frac{h}{2}) + f(t+h)) + O(h^5)$$

Predictor-Corrector Methods

The Predictor-Corrector is another popular family of algorithms for the numerical solution of ordinary differential equations. As the name implies, these are a family of itterative techniques, where you first predict the next step and then correct it using the new estimates.

We will introduce the general philosophy of predictor-corrector methods through a simple example. Consider the first order equation:

$$\dot{x}(t) = f(x(t),t) \textit{ with initial value } x(t_0) = x_0$$

Let us define

$$x_n = x(t_0 + nh)$$
$$\dot{x}_n = \dot{x}(t_0 + nh) = f(x_n, (t_0 + nh))$$

Using Simpson’s rule, we can write

$$x_{n+1} = x_{n-1} + \frac{h}{3}(\dot{x}_{n-1} + 4\dot{x}_n + \dot{x}_{n+1}) + O(h^5)$$

But from the defining state equations, we have:

$$\dot{x}_{n+1} = f(x_{n+1}, (t_0 + (n+1)h))$$

These two equations are solved itteratvely as predictor and corrector equations. However, to start the operation, we need an initial estimate of x_{n+1}. Milne's formula:

$$x_{n+1} = x_{n-3} + \frac{4h}{3}(2\dot{x}_n - \dot{x}_{n-1} + 2\dot{x}_{n-2}) + O(h^5)$$

may be used to obtain an initial value for x_{n+1} provided we have estimates for three previous values. Runga-Kutta method may be used to start the algorithm. The complete algorithm then is as follows:

Starting with x_0 at t_o, find x_1 , x_2 and x_3 at (t+h) and (t+2h) using the Runga Kutta algorithm.

Calculate $\dot{x}_1, \dot{x}_2 \; and \; \dot{x}_3$ using $\dot{x}_{n+1} = f(x_{n+1}, (t_0 + (n+1)h))$

Use Milne's formula $x_{n+1} = x_{n-3} + \frac{4h}{3}(2\dot{x}_n - \dot{x}_{n-1} + 2\dot{x}_{n-2}) + O(h^5)$ to obtain a starting value for x_4

Use the predictor-corrector pair of equations to refine the value of x_4

Use Milne's formula to obtain a starting value for the next step, refine using the predictor-corrector formulae, and repeat.

Finite difference and finite element methods

Ordinary differential equations (the type of equations we have encountered so far in circuit analysis and systems modelling) can be solved by transforming them into difference equations as follows:

$$\dot{x} = f(x, t)$$

is replaced by

$$\frac{\Delta x}{\Delta t} = f((x + \frac{\Delta x}{2}), (t + \frac{\Delta t}{2})),$$

where $(\Delta x, \Delta t)$ are the steps in the itteration process. This can be seen as a rather primitive version of the sophisticated algorithms such as Runga-Kutta that we have been studying, which only take into account first order terms. Neverthiless, it is a very efficient methos for the solution of ordinary differential equations.

When more than one independent variable is involved, we get partial differential equations (PDE) and the corresponding method is the finite element method.

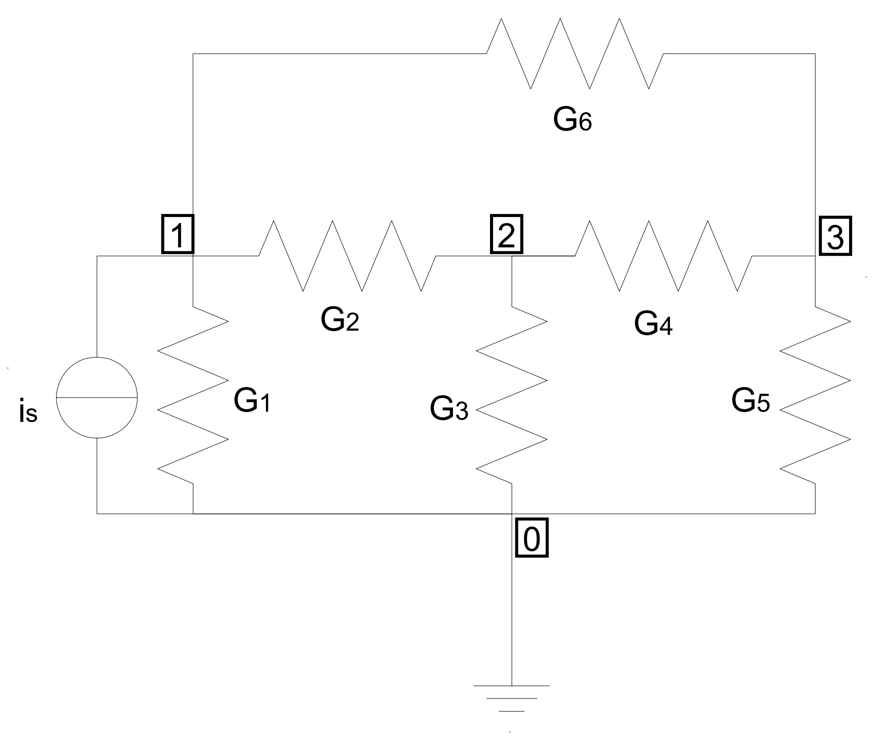
Typically, we encounter PDEs in problems associated with electromagnetic waves where the three space variables and time are all independent variables. PDEs also arise in other branches of engineering, in fluid flow, heat transfer and stree analysis, for example.

A treatment of the FEM will not be attempted here.

2.3.3 Networks with sparse matrices

Sparse matrices are generated in many engineering (and other) applications.

We will consider a few examples from arising from the formulation of circuit equations. Consider a circuit with four nodes as shown:



Writing the node equations with respect to the ground (node 0), we have:

$$\begin{bmatrix} G_1+G_2+G_6 & -G_2 & -G_6 \\ -G_2 & G_2+G_3+G_4 & -G_4 \\ -G_6 & -G_4 & G_4+G_5+G_6 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} i_s \\ 0 \\ 0 \end{bmatrix}$$

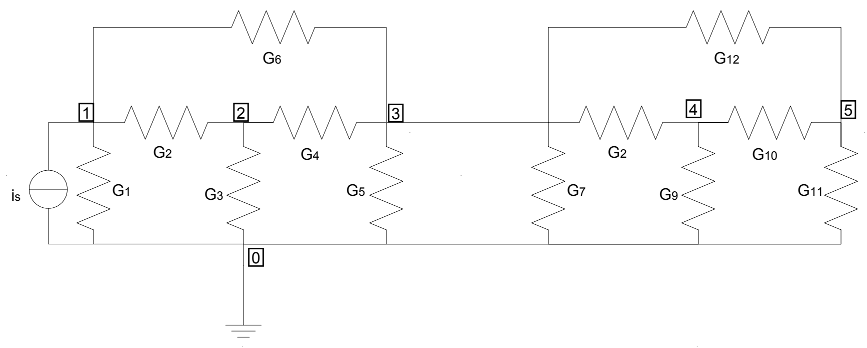
This can be written as:

$$Gv = i$$

Note that G is symmetric, and that the diagonal is probably dominant

Note also that this is not, sparse; it is in fact a full matrix.

Now let us look at a network formed by cascading two of these (except for the current source) as follows:



This has six nodes (including the reference node) and so five nodal equations, with 25 possible entries. However, we note that:

Node 1 is connected to only 2 other nodes,
Node 2 is connected to only 2 other nodes,
Node 3 is connected to only 4 other nodes,
Node 4 is connected to only 2 other nodes,
Node 5 is connected to only 2 other nodes,
so that the non-zero elements of the new conductance matrix are as indicated below:

$$\begin{bmatrix} * & * & * & & \\ * & * & * & & \\ * & * & * & * & * \\ & & * & * & * \\ & & * & * & * \end{bmatrix}$$

There are eight zero elements. Out of a total of 25. If we had another of the original networks connected in cascade, to give a 7 x 7 conductance matrix, we would have the following pattern:

$$\begin{bmatrix} * & * & * & & & & \\ * & * & * & & & & \\ * & * & * & * & * & & \\ & & * & * & * & & \\ & & * & * & * & * & * \\ & & & & * & * & * \\ & & & & * & * & * \end{bmatrix}$$

There are only twenty-five non-zero elements, out of a possible total of 49, that is the matrix is almost half empty. This of course is a particular example, but in general, as the size of the network increases, its sparsity also increases in most practical cases.

We define the sparsity of a matrix as the ratio between the number of zero elements and the total number of elements. In the last case, we have a sparsity of 24/49 = 0.49 or 49 %.

MATLAB has a number of demonstration matrices taken from real-life situations. The “west0479” is a matrix describing connections in a model of a diffusion column in a chemical plant. It is 479 x 479 and has 1887 non-zero elements.

The following instructions will load this matrix and set matrix A equal to it:

```
load west0479
A=west0479
```

We can examine its size and the number of non-zero elements using:

```
size(A)
          ans =

                    479   479

nnz(A)
          ans =

                         1887
```

Thus, the sparsity of this matrix is

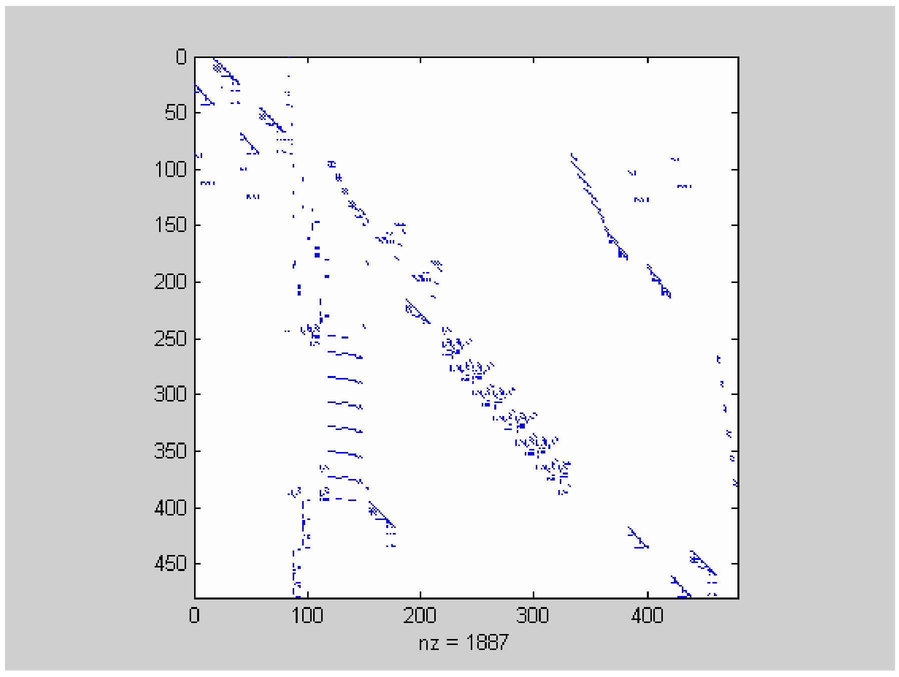
$$\frac{(479*479-1887)}{479*479}*100\%$$

Using MATLAB, we can obtain this as:

```
Per_cent_sparsity = 100*(prod(size(A))-nnz(A))/prod(size(A))

Per_cent_sparsity =

            99.1776
```

We can also obtain a plot of the positions where there are non-zero entries, similar to what we saw with the example network by using the MATLAB command “spy”:

```
spy(A)
```



Reordering for conservation of sparsity

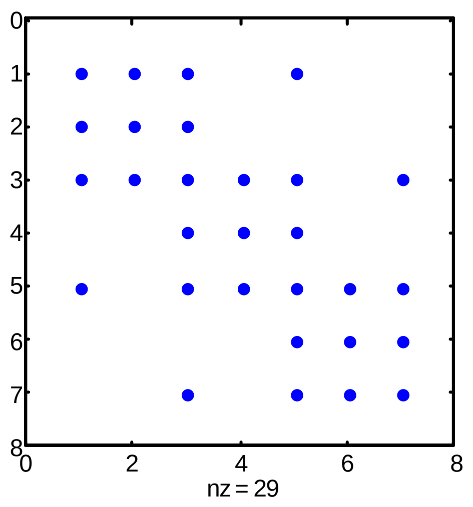
We have already looked at pivoting for reducing round-off errors, when considering Gaussian elimination. In addition to ensuring that the diagonal element be non-zero (a zero diagonal element will lead to a breakdown of the process), it is better that it be comparatively large, as this would reduce computational errors.

We will now look at the special case of sparse matrices, where it is desirable to maintain sparsity during he process of factorisation.

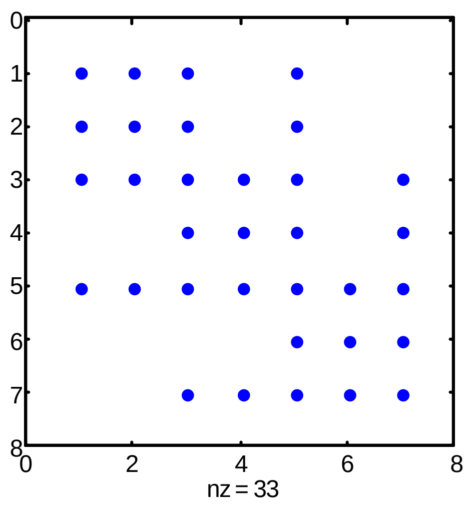
We saw that the processing of the sparse matrix considered in LU factorisation did not result in adding new non-zero elements. This is not always so. We will consider a slightly modified matrix to illustrate this point.

$$\begin{bmatrix} 5 & -2 & -2 & 0 & -1 & 0 & 0 \\ -2 & 5 & -2 & 0 & 0 & 0 & 0 \\ -2 & -2 & 11 & -2 & -2 & 0 & -1 \\ 0 & 0 & -2 & 5 & -2 & 0 & 0 \\ -1 & 0 & -2 & -2 & 11 & -2 & -2 \\ 0 & 0 & 0 & 0 & -2 & 5 & -2 \\ 0 & 0 & -1 & 0 & -2 & -2 & 5 \end{bmatrix}$$

The non-zero elements of this matrix is shown below:



The non-zero elements of the matrix after LU factorisation (both lower and upper triangles entered on one matrix, with implied unity elements on the diagonal) obtained using MATLAB is as shown:



Note that there are four additional non-zero elements, which have arisen as a result of the factorisation.

We need to look at the possibility of reducing the addition of new elements, by proper ordering of the equations. MATLAB has two important reordering schemes:

Reverse-Cuthill-McKee reordering scheme
Symmetric Minimum Degree scheme
They are described as follows:

SYMRCM Symmetric reverse Cuthill-McKee permutation.

p = SYMRCM(S) returns a permutation vector p such that S(p,p) tends to have its diagonal elements closer to the diagonal than S. This is a good preordering for LU or Cholesky factorization of matrices that come from "long, skinny" problems. It works for both symmetric and asymmetric S.

SYMMMD Symmetric minimum degree permutation.

p = SYMMMD(S), for a symmetric positive definite matrix S, returns the permutation vector p such that S(p,p) tends to have a sparser Cholesky factor than S. Sometimes SYMMMD works well for symmetric indefinite matrices too.

They both give “better” results with the LU factorisation than the original, in that the number of non-zero elements introduced is reduced.

However, the most obvious and the simplest reordering scheme is to order the rows (and columns) in increasing number of non-zero elements. In this particular case, it yields the order:

2 4 6 1 7 3 5

When the rows and columns are reordered in this manner (so that the diagonal elements remain as diagonal elements), the new matrix is:

```
A =

    5   -2   -2    0   -1    0    0
   -2    5   -2    0    0    0    0
   -2   -2   11   -2   -2    0   -1
    0    0   -2    5   -2    0    0
   -1    0   -2   -2   11   -2   -2
    0    0    0    0   -2    5   -2
    0    0   -1    0   -2   -2    5
```

We will compare these three reordering schemes, with respect to our example.

```
» p=symrcm(A)

p =

    2    1    7    6    3    5    4

» q=symmmd(A)

q =

    4    1    2    6    7    3    5

» r

r =

    2    4    6    1    7    3    5

» A1=A(p,p)

A1 =

    5   -2    0    0   -2    0    0
   -2    5    0    0   -2   -1    0
    0    0    5   -2   -1   -2    0
    0    0   -2    5    0   -2    0
   -2   -2   -1    0   11   -2   -2
    0   -1   -2   -2   -2   11   -2
    0    0    0    0   -2   -2    5

» A2=A(q,q)
```

```
A2 =

    5    0    0    0    0   -2   -2
    0    5   -2    0    0   -2   -1
    0   -2    5    0    0   -2    0
    0    0    0    5   -2    0   -2
    0    0    0   -2    5   -1   -2
   -2   -2   -2    0   -1   11   -2
   -2   -1    0   -2   -2   -2   11

» A3=A(r,r)

A3 =

    5    0    0   -2    0   -2    0
    0    5    0    0    0   -2   -2
    0    0    5    0   -2    0   -2
   -2    0    0    5    0   -2   -1
    0    0   -2    0    5   -1   -2
   -2   -2    0   -2   -1   11   -2
    0   -2   -2   -1   -2   -2   11

» nnz(A)

ans =

   29

» nnz(lu(A))

ans =

   33

» nnz(lu(A1))

ans =

   31

» nnz(lu(A2))

ans =

   31

» nnz(lu(A3))

ans =

   29
```

The fact that the simple rank-order reordering is the best in this case (as it does not introduce any new non-zero) elements does not mean that it is always the best. It is very much dependant on the structure of the matrix under consideration.

Intuitively, a better scheme would be to reorder the balance equations after each row is processed, in the order of the freshly computed rank order. This is much more time consuming, but would be justified if the factored matrix is to be repeatedly used with new vectors (b), as is the case with (say) power system load flow studies. A still better algorithm is to allow for the fact that some of the original non-zero elements may actually vanish during processing due to cancellation, and to determine the rank order at each stage, taking into account such cancellations. This is even more time consuming than the previous method, but may be justified under special circumstances, such as in repeated on-line transient analysis.

Sparsity programming

The efficient storage and retrieval of sparse matrices need special programming techniques, if we are to exploit their sparsity. We can reduce both storage and computational requirements for the processing of such matrices by proper choice of techniques. Some reservations have been expressed in recent times about some of the traditional methods used, on account of the relative burdens of computation and access times of modern personal computers. It has also been pointed out that storage is now comparatively cheap. However, along with the advancement of technology that has brought cheap mass storage, the dimensions of the problems that need to be tackled has also increased. Therefore, there is a continuing need for good and efficient programming methods for the handling of very large sparse matrices.

MATLAB has a special collection of routines for handling sparse matrices. We have already used some of them, without bothering about how such matrices are stored.

Let us consider our continuing example.

```
» A

A =

   5   -2   -2    0   -1    0    0
  -2    5   -2    0    0    0    0
  -2   -2   11   -2   -2    0   -1
   0    0   -2    5   -2    0    0
  -1    0   -2   -2   11   -2   -2
   0    0    0    0   -2    5   -2
   0    0   -1    0   -2   -2    5

» sparse(A)

ans =
   (1,1)        5
   (2,1)       -2
   (3,1)       -2
   (5,1)       -1
   (1,2)       -2
   (2,2)        5
   (3,2)       -2
   (1,3)       -2
   (2,3)       -2
   (3,3)       11
   (4,3)       -2
   (5,3)       -2
   (7,3)       -1
   (3,4)       -2
   (4,4)        5
   (5,4)       -2
   (1,5)       -1
   (3,5)       -2
   (4,5)       -2
   (5,5)       11
   (6,5)       -2
   (7,5)       -2
   (5,6)       -2
   (6,6)        5
   (7,6)       -2
   (3,7)       -1
   (5,7)       -2
   (6,7)       -2
   (7,7)        5
```

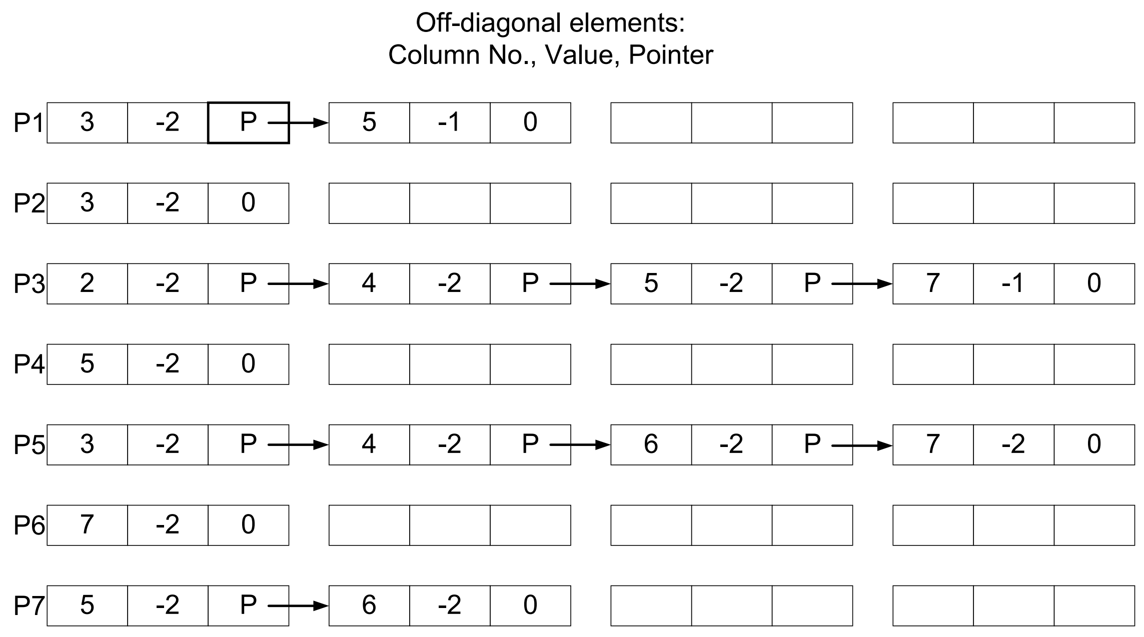
The instruction sparse (A) has converted the storage of the matrix A from its normal form into the sparse matrix representation in MATLAB. As can be seen, this representation uses two integer arrays to indicate the indices of each non-zero element and another real (or complex) array to represent the value of each element. In the case of this example, it is obviously not an efficient mode of storage, for we have used a total of (3*29 = 87) locations to store 49 (including zero) elements. However, it comes to its own as the size of the matrix and the sparsity increases, as in the case of the test matrix presented earlier.

We will now consider a slightly more sophisticated mode of representation related to this same method, which allows for the fact that the diagonal element of most matrices of practical interest would be non-zero, and also facilitates easy reordering.

The first column gives the values of the diagonal elements, in order, as at the beginning. The second and the third columns give the row ordering scheme, and at the start, it is simply 1, 2,3 etc. We will later see why we need two columns.

Diagonal elements	Row pointers and reverse pointers		Off-diagonal elements: Column No., Value, Pointer		
5	1	1	2	-2	P1
5	2	2	1	-2	P2
11	3	3	1	-2	P3
5	4	4	3	-2	P4
11	5	5	1	-1	P5
5	6	6	5	-2	P6
5	7	7	3	-1	P7

The next set of three columns give the first non-zero off-diagonal element in each row as a combination of three values The first of these give the column index, the second gives the element value and the third is a pointer to the location of the next non-zero element in the row. The pointer will be set to zero if there are no more non-zero values.


Off-diagonal elements:
Column No., Value, Pointer
P1 3 -2 P → 5 -1 0
P2 3 -2 0
P3 2 -2 P → 4 -2 P → 5 -2 P → 7 -1 0
P4 5 -2 0
P5 3 -2 P → 4 -2 P → 6 -2 P → 7 -2 0
P6 7 -2 0
P7 5 -2 P → 6 -2 0

If we now reorder the equations according to (say) the pattern r discussed earlier, we will not move any of the values other than the pointers and reverse pointers on the second and third columns:

r = 2 4 6 1 7 3 5

Diagonal elements	Row pointers and reverse pointers		Off-diagonal elements: Column No., Value, Pointer		
5	4	2	2	-2	P1
5	1	4	1	-2	P2
11	6	6	1	-2	P3
5	2	1	3	-2	P4
11	7	7	1	-1	P5
5	3	3	5	-2	P6
5	5	5	3	-1	P7

The original matrix and the reordered matrix are as follows:

```
A =

   5   -2   -2    0   -1    0    0
  -2    5   -2    0    0    0    0
  -2   -2   11   -2   -2    0   -1
   0    0   -2    5   -2    0    0
   0    0   -2   -2   11   -2   -2
   0    0    0    0   -2    5   -2
   0    0   -1    0   -2   -2    5

>> r=[2 4 6 1 7 3 5]

r =

   2    4    6    1    7    3    5

>> A1=A(r,r)

A1 =

   5    0    0   -2    0   -2    0
   0    5    0    0    0   -2   -2
   0    0    5    0   -2    0   -2
  -2    0    0    5    0   -2   -1
   0    0   -2    0    5   -1   -2
  -2   -2    0   -2   -1   11   -2
   0   -2   -2    0   -2   -2   11
```

Follow the pointers and work out how the indices help you to interpret the entroes after reordering.