2

State and state space

We have been studying electrical circuits with reference to the complex frequency s and the s-plane. You have seen this to be a technique that is most convenient, especially when you are interested in the steady-state behaviour of a circuit. There is an alternative way in which we can model dynamic systems, including electrical circuits, which is more suited for the study of their transient behaviour.

The state of a system in this context means exactly the same as the common English meaning of the word. However, we need to be able to define it more precisely. For a known system, a knowledge of its present state and any future inputs from the external world should be sufficient for us to be able to predict its future state. This then is what the state means. The variables, whose values tell us what the present state is, are known as state variables.

A set of state variable is the minimum set of variables, whose values at any instant, along with the inputs from then onwards, would enable the complete determination of the values of the state variables in the future. In effect, the state variables contain all the information about the past and the present, necessary for the prediction of their future values.

2.1 State variables and state space representation

Introduction

We are used to the description of dynamic systems using differential equations. We will examine their relationship through a simple example.

A pendulum, swinging (with small amplitude) in a gravitational field, in a vacuum will continue to swing indefinitely as there is no resistance to slow it down. Similarly, an initially charged capacitor shorted through an inductor (assumed to be without resistance) will give rise to a cyclic charge-discharge activity. In the absence of any circuit resistance or leakage across the capacitor, this will continue indefinitely. Both these phenomena may be described by a simple second order differential equation.

$$d^2 x / dt^2 + \omega_n^2 x = 0$$

[If there were any resistance, the equation would be of the form
:

$$d^2 x / dt^2 + 2\zeta\omega_n \, dx / dt + \omega_n^2 x = 0 \quad]$$

We can break this down into two simultaneous first order differential equations by defining two new variables:

$$x_1 = x$$
$$x_2 = dx / dt = dx_1 /dt$$

Then,

$$dx_1 / dt = x_2$$

$$dx_2 /dt = -\omega_n^2 x_1$$

If we know the values of x_1 and x_2 at any instant of time t_1, then, it is possible to compute their values for all $t > t_1$ (We will also need to know all inputs for $t \geq t_1$., assumed to be zero in the above formulation) Thus, x_1 and x_2 are a valid set of state variables. They are of course not the only valid set, for, we could define (say)

$$z_1 = x_1 + x_2$$
$$z_2 = x_1 – x_2,$$

Then, we can obtain x_1 and x_2 from z_1 and z_2, and hence, (z_1, z_2) is also a valid set of state variables. There are thus an infinite number of such sets of state variables.

The space defined by the state variables is known as the state-space. In the above example, the two-dimensional space x_1-x_2 is the state-space, and any point on it will represent a state of the system. If the state vector in a 3-vector, then its corresponding state-space is also three-dimensional. An n-dimensional state vector will describe a motion in an n-dimensional state-space.

We saw how the state space representation relates to the description of a physical system otherwise described by a set of differential equations. We will now examine the relationship between state space and the system function of a network. The system function is, as we saw earlier, a function of the complex frequency s, representing the ratio of the Laplace transform of a response to the Laplace transform of the excitation causing the response. In this sense, we can look at all such functions as transfer functions.

We have already seen that if the (network) model can be arranged as a set of interconnections among integrators, their outputs constitute a possible set of state variables. With this background, we will attempt to breakdown a function of the complex variable s into a set of relations that can be easily represented by integrators.

We have also noted that the most general form of a system function is given by a *real rational function of s*, expressed as the quotient of two polynomials in s.

$H(s) = q(s) / p(s)$, where

$$q(s) = b_m(s-z_1)(s-z_2) \;.\;.\;.\; (s-z_m)$$
$$p(s) = (s-p_1)(s-p_2) \;.\;.\;.\;.\; (s-p_n)$$

Unlike in the study of other more general systems (such as in Control Systems), we have the advantage that network functions of passive networks are subject to certain constraints. The poles and zeros are simple, that is there are no higher order poles or zeros, and there are no poles or zeros on the right hand side of the s-plane.

We can expand H(s) into partial fractions, so that each term corresponds to a single integrator, whose output can then be considered as a state variable. We will, for completeness, examine the general case of how to obtain a set of state variables, given a real rational function of s.

This is of course a very mechanical treatment, giving no physical insight into the problem being studied. It is much more useful to recognise suitable physical quantities as the state variables, and obtain the equations governing their relationships.

We saw that there are many alternate ways of describing the behaviour of a dynamic system, using different state variables. They are all transformations of each other, and most have no physical significance. In the case of the simple pendulum considered earlier, we could have derived the state space description using the angular displacement and the angular velocity as state variables. These obviously have physical significance. But an equally valid, from a representational point of view, pair of state variables would have been the sum of the angular displacement and the angular velocity as one variable and their difference as the other. This combination does not make physical sense.

In engineering, we would always prefer to formulate our equations in terms of variables that have some physical significance. This has a number of advantages, including the ability to make reality checks and providing us with an insight into the problem under study.

We noted earlier on that the state variables contain all the information about the past and the present necessary for the prediction of future behaviour. There are many ways in which these can be visualised. They are connected with "memory", and they are in some manner associated with energy storage. In linear electrical circuits, only inductors and capacitors can store energy (Resistors dissipate energy, but cannot store it.)

We have a choice in the selection of state variables with physical significance. We could select either (flux ϕ, charge q) or (current i, voltage v) as our preferred set of variables. Other combination including different combinations of these are of course possible as we saw earlier, but these seem to offer alternatives with real physical significance.

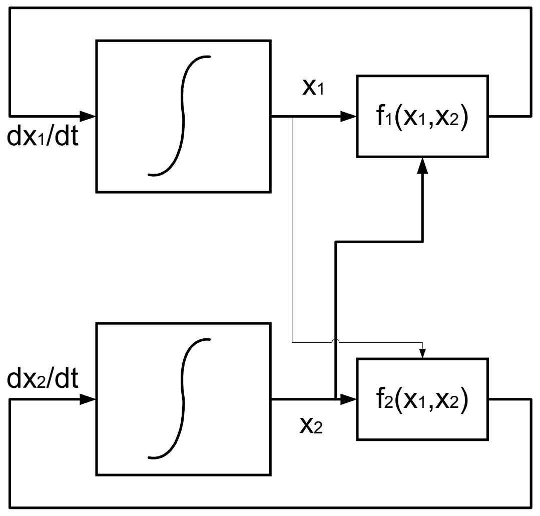
We may use the methods of nodal and mesh analysis, or alternatively, energy function methods to obtain the state equations of a system.

2.1.1 State-space

A second order system (with no inputs) may be represented by a set of two state equations as follows:

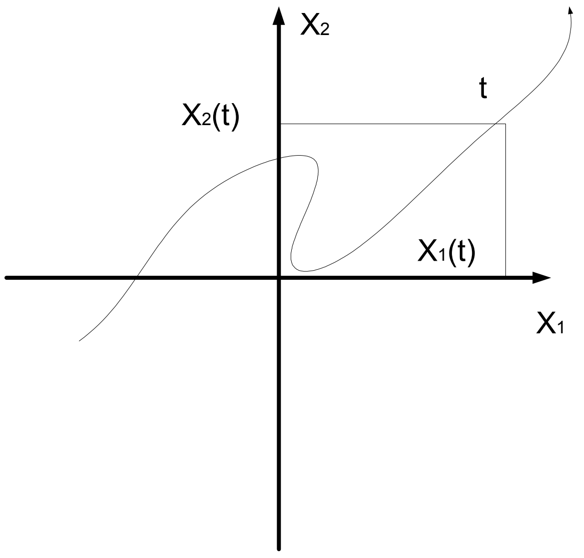
$$\dot{x}_1 = f_1(x_1, x_2)$$
$$\dot{x}_2 = f_2(x_1, x_2)$$

We can draw a block diagram of these two equations, using two integrators, as follows:



The outputs of the integrators may be taken as a possible set of state variables.

We can then visualise this system as moving in a two-dimensional space as shown below.



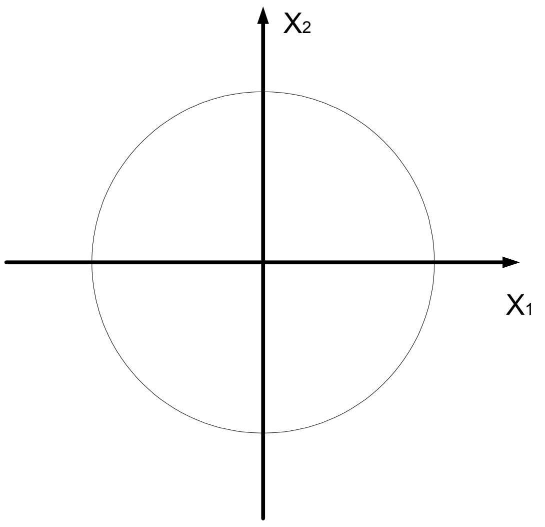
A point [x_1(t), x_2(t)] on the path represents the state of the system at time t. It is obvious that the system shown above would be non-linear, and be a rather complex one.

The simple harmonic motion represented by the set of linear equations

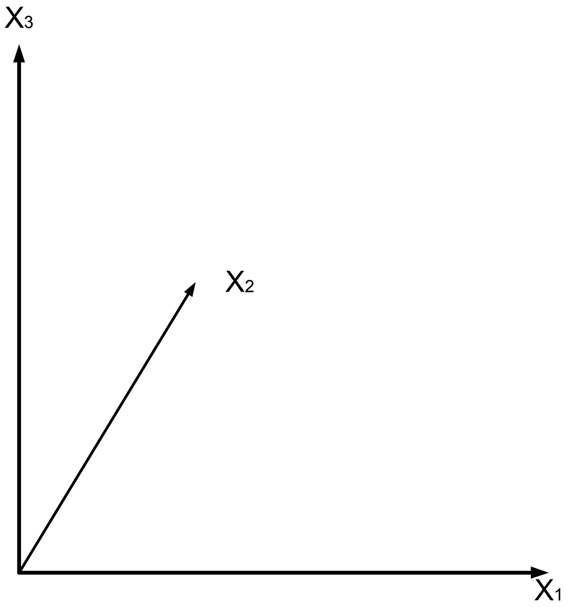
$$dx_1 / dt = x_2$$

$$dx_2 /dt = -\omega_n^2 x_1$$

considered earlier would be represented on the state-space as follows:



A trajectory on a three-dimensional state-space may be visualised as follows:



Unfortunately, it is not possible to directly represent spaces of higher dimensions on a two-dimensional surface, but you should not have much difficulty in visualising the extension of the concept of state – space to n dimensions.

2.1.2 Obtaining state variables and state equations from a transfer function

Consider the general form of a transfer function:

$$H(s) = \frac{b_n s^n + b_{n-1} s^{n-1} + \ . \ . \ . + b_1 s + b_0}{s^n + a_{n-1} s^{n-1} + \ . \ . \ . + a_1 s + a_0}$$

Let the function H(s) denote a relationship between two functions U(s) and Y(s) such that:

$$\frac{Y(s)}{U(s)} = H(s)$$

$$Y(s)[s^n + a_{n-1}s^{n-1} +. \ . \ . + a_1 s + a_0] = U(s)[b_n s^n + b_{n-1}s^{n-1} + \ . \ . \ . + b_1 s + b_0]$$

Dividing by s^n and rearranging,

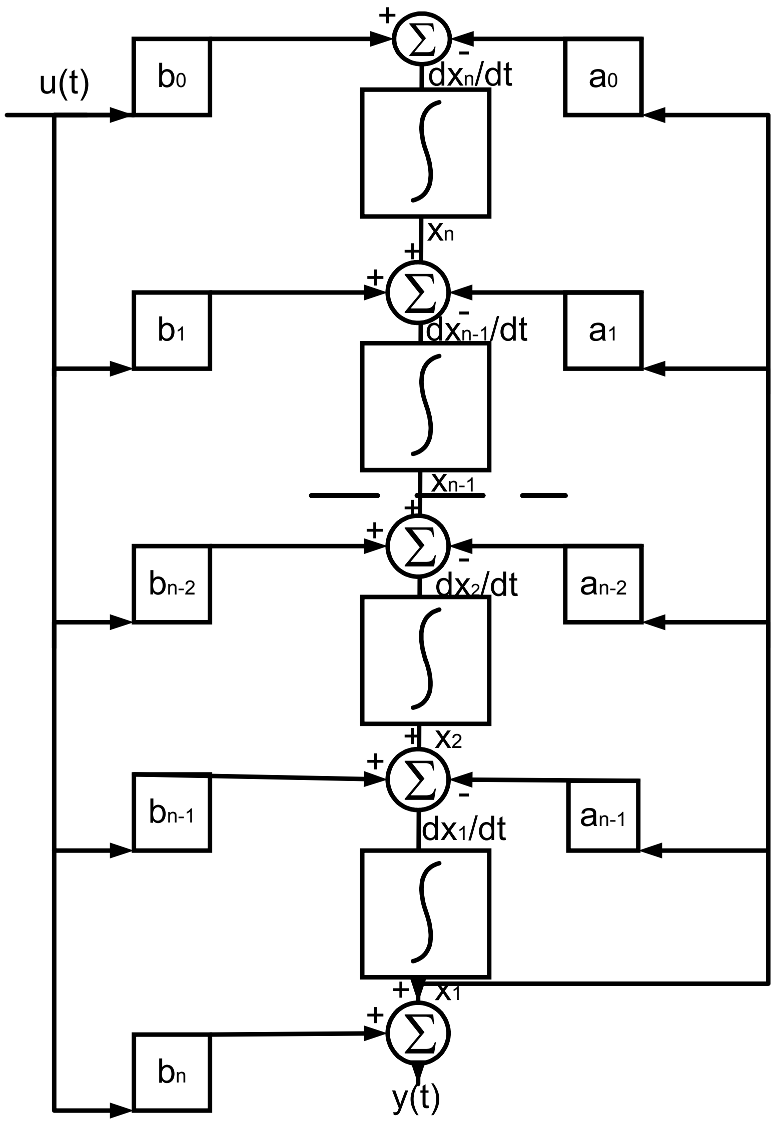
$$Y(s) = b_n U(s) + \frac{1}{s}[b_{n-1}U(s) - a_{n-1}Y(s)] + \frac{1}{s^2}[b_{n-2}U(s) - a_{n-2}Y(s)] +$$

$$+ \ . \ . \ . \ . \ +$$

$$\frac{1}{s^{n-1}}[b_1 U(s) - a_1 Y(s)] + \frac{1}{s^n}[b_0 U(s) - a_0 Y(s)]$$

The implementation of these using integrators is shown in the figure.

From this, we can write down the state space description by inspection, if we chose the outputs of the integrators as the state variables.



$$\dot{x}_1 = -a_{n-1}x_1 + x_2 + b_{n-1}u$$
$$\dot{x}_2 = -a_{n-2}x_1 + x_3 + b_{n-2}u$$
$$. \quad . \quad . \quad .$$
$$\dot{x}_{n-1} = -a_1 x_1 + x_n + b_1 u$$
$$\dot{x}_n = -a_0 x_1 + b_0 u$$

The output y is given by:

$$y = x_1 + b_0 u$$

In matrix form, we can write:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ . \\ . \\ \dot{x}_{n-1} \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} -a_{n-1} & 1 & 0 & . & . & . & 0 \\ -a_{n-2} & 0 & 1 & 0 & . & . & 0 \\ . & & & & & & \\ . & & & & & & \\ -a_1 & 0 & 0 & 0. & . & . & 1 \\ -a_0 & 0 & 0 & 0 & . & . & 0. \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} b_{n-1} \\ b_{n-2} \\ . \\ . \\ b_1 \\ b_0 \end{bmatrix} u$$

$$[y] = [1 \quad 0 \quad 0 \quad . \quad . \quad . \quad 0] \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ .x_{n-1} \\ x_n \end{bmatrix} + b_n u$$

In concise form:

$$\dot{X} = AX + BU$$
$$Y = CX + DU$$

As has been repeatedly emphasised, there is no unique set of state variables, and what we defined in the above analysis is just one possible selection of state variables, given the function H(s). We will now examine one other possible form, obtained by the partial fraction expansion of the given function.

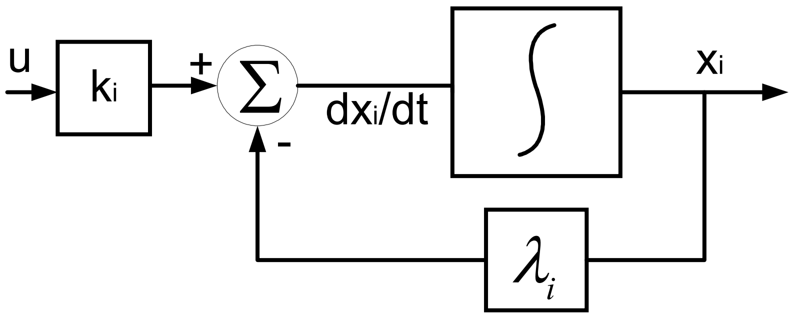
Let us first assume that all the poles are distinct and real. Then, we can write:

$$H(s)=\frac{Y(s)}{U(s)}=\sum_{i=1}^{n}\frac{k_i}{s+\lambda_i}$$

Considering each term,

$$\frac{X_i(s)}{U(s)}=\frac{k_i}{s+\lambda_i}$$
$$sX_i(s)=k_iU(s)-\lambda_iX(s)$$
$$X_i(s)=\frac{1}{s}\left[k_iU(s)-\lambda_iX(s)\right]$$

This leads to:



If we again select the outputs of the integrators as the state variables:

$$\dot{x}_i=-\lambda_ix_i+k_iu$$
$$y=\sum_{i=1}^{n}x_i$$

In matrix form:

$$\begin{bmatrix}\dot{x}_1\\ \dot{x}_2\\ .\\ .\\ \dot{x}_{n-1}\\ \dot{x}_n\end{bmatrix}=\begin{bmatrix}-\lambda_1 & 0 & 0 & . & . & . & 0\\ 0 & -\lambda_2 & 0 & 0 & . & . & 0\\ . & & & & & & \\ . & & & & & & \\ 0 & 0 & 0 & . & . & -\lambda_{n-1} & 0\\ 0 & 0 & 0 & . & . & . & -\lambda_n\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ .\\ .\\ x_{n-1}\\ x_n\end{bmatrix}+\begin{bmatrix}k_1\\ k_2\\ .\\ .\\ k_{n-1}\\ k_n\end{bmatrix}u$$

$$[y] = [1 \quad 1 \quad 1 \quad . \quad . \quad . \quad 1]\begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ .x_{n-1} \\ x_n \end{bmatrix}$$

This is again of the form

$$\dot{X} = AX + BU$$
$$Y = CX + DU$$

even though the structure of the matrices are different.

2.1.3 Resistors, inductors and capacitors

Consider the following circuit elements:

Resistance R (ohms)
Conductance G (mhos) (=1/R)

Capacitance C (farads)
Elastance S (darafs) (=1/C)

Inductance L (henrys)
Inverse inductance Γ
(inverse henrys) (=1/L)

For the class of linear elements, we have the following relationships:

Defining relationships:

$$v = Ri, \qquad i = Gv$$
$$q = Cv, \qquad v = Sq$$
$$\phi = Li, \qquad i = \Gamma\phi$$

i-v relationships:

$$v = Ri, \qquad i = Gv$$

$$i = C\frac{dv}{dt}, \qquad \frac{dv}{dt} = Si$$

$$v = L\frac{di}{dt}, \qquad \frac{di}{dt} = \Gamma v$$

The relationships for capacitors and inductors may also be written in integral form as:

$$v(t) = \frac{1}{C}\int_{t_0}^{t} i(\tau)d\tau + v_0$$

$$i(t) = \frac{1}{L}\int_{t_0}^{t} v(\tau)d\tau + i_0$$

Relationships derived from resistances do not yield state equations, as they can only be algebraic equations and not differential equations. If we select v-i relationships as the relationships of choice (this is not necessary, we could equally well select others such as flux – charge relationships), we are led naturally to select:

voltages across capacitors

and

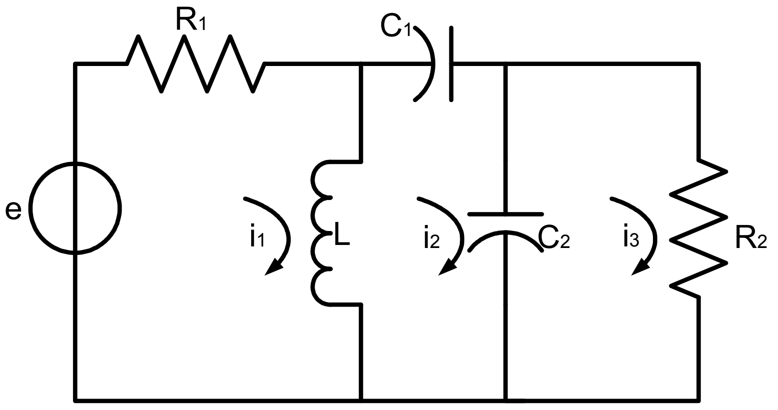
currents through inductors

as the state variables of choice.

[In the generalised theory of dynamic systems, there are analogous choices to be made for the representation of physical phenomena in mechanical systems, thermodynamic systems and fluidic systems]

2.1.4 Formulation of state equations by nodal and mesh analysis

We will demonstrate the derivation of state equations through nodal and mesh analysis through a simple example. Consider the network shown below:



Writing the mesh equations:

$$R_1 i_1 + L\dot{i}_1 - L\dot{i}_2 = e$$
$$-L\dot{i}_1 + L\dot{i}_2 + \frac{1}{C_1}\int i_2 dt + \frac{1}{C_2}\int i_2 dt - \frac{1}{C_2}\int i_3 dt = 0$$
$$-\frac{1}{C_2}\int i_2 dt + \frac{1}{C_2}\int i_3 dt + R_2 i_3 = 0$$

Differentiating the second and third equations to transform from integral to differential form, these may be rewritten as:

$$R_1 i_1 + L\dot{i}_1 - L\dot{i}_2 = e$$
$$-LC_1C_2\ddot{i}_1 + LC_1C_2\ddot{i}_2 + (C_1 + C_2)i_2 - C_1 i_3 = 0$$
$$-i_2 + i_3 + C_2R_2\dot{i}_3 = 0$$

From the above, it appears that there are two variables with second derivatives. In fact, there is only one, (i_1-i_2). We need to recognise this fact, that there is only one variable with a second derivative.

If we rewrite the equations using (i_1-i_2) as a single variable (and also abandoning (say) i_1 as an independent variable):

$$R_1(i_1 - i_2) + R_1 i_2 + L\overbrace{(i_1 - i_2)}^{\bullet} = e$$
$$-LC_1C_2\overbrace{(i_1 - i_2)}^{\bullet\bullet} + (C_1 + C_2)i_2 - C_1 i_3 = 0$$
$$-i_2 + i_3 + C_2R_2\dot{i}_3 = 0$$

As there are no derivatives of i_2, it can be eliminated (by substituting for i_2 from the third equation, into the other two) to yield:

$$R_1(i_1 - i_2) + L\overbrace{(i_1 - i_2)}^{\bullet} + R_1 i_3 + C_2R_1R_2\dot{i}_3 = e$$
$$-LC_1C_2\overbrace{(i_1 - i_2)}^{\bullet\bullet} + (C_1 + C_2)(i_3 + C_2R_2\dot{i}_3) - C_1 i_3 = 0$$

If we now define:

$$x_1 = i_1 - i_2$$
$$x_2 = \overbrace{(i_1 - i_2)}^{\bullet}$$
$$x_3 = i_3$$

We get the state equations:

$$\begin{aligned}
&\dot{x}_1 = x_2 \\
&R_1x_1 + Lx_2 + R_1x_3 \_ C_2R_1R_2\dot{x}_3 = e \\
&-LC_1C_2\dot{x}_2 + (C_1 + C_2)C_2R_2\dot{x}_3 + C_2x_3 = 0
\end{aligned}$$

Rearranging, we get:

$$\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_3 &= \frac{1}{C_2R_2}x_1 + \frac{L}{C_2R_1R_2}x_2 + \frac{1}{C_2R_2}x_3 - \frac{1}{C_2R_1R_2}e \\
\dot{x}_2 &= \frac{1}{LC_1C_2}\left[(C_1 + C_2)C_2R_2\dot{x}_3 + C_2x_3\right] \\
&= \frac{(C_1 + C_2)R_2}{LC_1}\left[\frac{1}{C_2R_2}x_1 + \frac{L}{C_2R_1R_2}x_2 + \frac{1}{C_2R_2}x_3 - \frac{1}{C_2R_1R_2}e\right] + \frac{1}{LC_1}x_3 \\
&= \frac{C_1 + C_2}{LC_1C_2}x_1 + \frac{C_1 + C_2}{C_1C_2R_1}x_2 + \frac{C_1 + 2C_2}{LC_1C_2}x_3 - \frac{C_1 + C_2}{LC_1C_2R_1}e
\end{aligned}$$

In matrix form,

$$\dot{X} = AX + BU$$

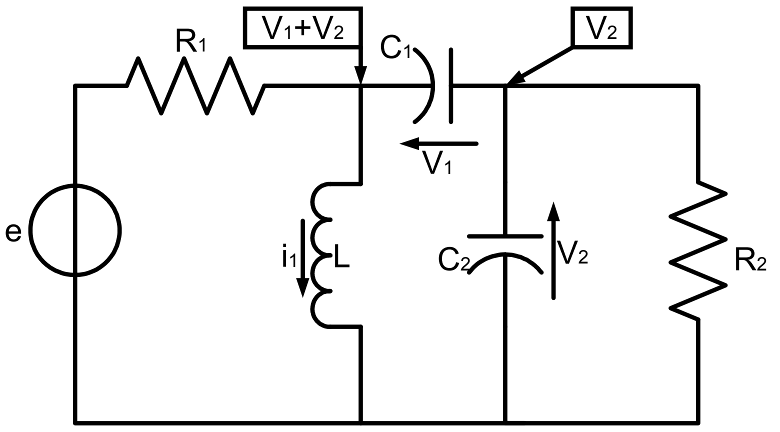
where:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \qquad A = \begin{bmatrix} 0 & 1 & 0 \\ \dfrac{C_1 + C_2}{LC_1C2} & \dfrac{C_1 + C}{C_1C_2R_1} & \dfrac{C_1 + 2C_2}{LC_1C_2} \\ \dfrac{1}{C_2R_2} & \dfrac{L}{C_2R_1R_2} & \dfrac{1}{C_2R_2} \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ -\dfrac{C_1 + C_2}{LC_1C_2R_1} \\ -\dfrac{1}{C_2R_1R_2} \end{bmatrix}$$

We could, alternatively, have started by writing the nodal equations, and obtained the state equations in terms of the node-pair voltages and their derivatives.

Another approach (to both nodal and mesh analysis) is to define the state variables in terms of physical variables that correspond directly to the state – the currents through inductors and voltages across capacitors. This would be the most direct approach, and we will repeat the analysis of this circuit using his approach. (This is a method that can be carried across to the study of other systems such as mechanical, thermal and fluid systems, which all have identifiable "through" variables and "across" variables)



Writing the nodal equations in terms of i$_1$, v$_1$ and v$_2$, we have:

$$\frac{v_1 + v_2 - e}{R_1} + i_1 + C_1 \dot{v}_1 = 0$$

$$-C_1 \dot{v}_1 + C_2 \dot{v}_2 + \frac{v_2}{R_2} = 0$$

We also have the additional relationship:

$$v_1 + v_2 = L\dot{i}_1$$

Rearranging, we have:

$$\dot{i}_1 = \frac{1}{L} v_1 + \frac{1}{L} v_2$$

$$\dot{v}_1 = -\frac{1}{C_1} i_1 - \frac{1}{R_1 C_1} v_1 - \frac{1}{R_1 C_1} v_2 + \frac{1`}{R_1 C_1} e$$

$$\dot{v}_2 = -\frac{1}{C_2} i_1 - \frac{1}{R_1 C_2} v_1 - \frac{R_1 + R_2}{C_2 R_1 R_2} v_2 + \frac{1}{R_1 C_2} e$$

This is in the standard form. Note how easily the state equations could be obtained, if we make a proper choice of state variables.

We could equally well have obtained the state equations by writing the mesh equations.

We can now formally state the procedure for writing down the state equations as follows:

1. Select the currents through inductors and voltages across capacitors as the state variables.
2. Write the loop *(node-pair)* equations for all loops *(node-pairs)* that contain *(are connected to)* at least one storage element (that is, an inductor or capacitor)
3. If there are n storage elements and only m (m<n) loop (node-pair) equations, then there will be an additional (n-m) relationships between the variables we have chosen. Altogether, there will be n equations.

2.1.5 Energy functions

We have defined the loop-based energy functions as:

$$T = \bar{I}^T L I = \sum_{i,k=1}^{l} L_{ik} I_i \bar{I}_k$$

$$F = \bar{I}^T R I = \sum_{i,k=1}^{l} R_{ik} I_i \bar{I}_k$$

$$V = \bar{I}^T S I = \sum_{i,k=1}^{l} S_{ik} I_i \bar{I}_k$$

where l is the number of independent loops and the node-pair-based energy functions as:

$$V^* = \bar{E}^T C E = \sum_{i,k=1}^{n} C_{ik} E_i \bar{E}_k$$

$$F^* = \bar{E}^T G E = \sum_{i,k=1}^{n} G_{ik} E_i \bar{E}_k$$

$$T^* = \bar{E}^T \Gamma E = \sum_{i,k=1}^{n} \Gamma_{ik} E_i \bar{E}_k$$

where n is the number of independent node-pairs.

L, R and S represent the loop inductance, resistance and elastance (reciprocal capacitance) while C,G and Γ represent the node capacitance, conductance and reciprocal inductance matrices.

Each of these energy functions is a positive semi-definite quadratic form. Their positive-semi-definiteness may be established qualitatively by considering that the loop and node-pair matrices in a passive network have to be positive-semi-definite because in such a network, the energy stored or dissipated cannot be negative.

Considering the energy functions derived from the loop equations, if we set

$$I_i = 0,\ \ i \neq 1$$
$$I_1 = 1$$

then we get:

$$Z_{11} = sT + F + V/s$$

Similarly, starting with the energy functions defined using node-pair equations and setting

$$E_i = 0, \qquad\qquad i \neq 1$$
$$E_1 = 1$$

we get

$$Y_{11} = sV^* + F^* + T^*/s$$

We may derive special cases from these results. For example, for an LC network, F is zero and for an RC network, T is zero. Therefore:

$$Z_{11} = sT + V/s \qquad\qquad \textit{for an LC network}$$
$$Z_{11} = F + V/s \qquad\qquad \textit{for a RC network}$$

Similar results may be obtained for the driving point admittances using the node-pair-based energy functions. Obviously, these are not the methods used for evaluating driving point impedances and admittances, as they lead to complex expressions. However, the insight provided by this analysis is very useful.

2.1.6 Formulation of state equations using energy functions

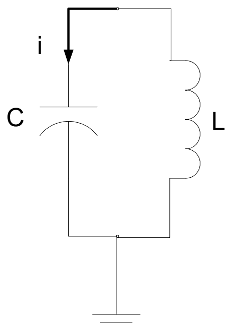
We have already briefly come across the use of energy functions in the previous lecture, where we referred to the modelling of the capacitor microphone. Let us now look at it in a little more detail.

Conservative systems

We will first consider conservative systems, that is, systems without energy sources or sinks, and later go on to consider non-conservative systems.

Example 1

Consider a very simple example, with only one inductor and capacitor as shown.



Let us define T as the total system kinetic energy and V as the total system potential energy. We will call the total energy E

In terms of the charge q and its derivative, we can write:

$$T = \frac{1}{2}L\dot{q}^2$$

$$V = \frac{1}{2C}q^2$$

$$E = T + V = \frac{1}{2}L\dot{q}^2 + \frac{1}{2C}q^2$$

Since the system is conservative,

$$\frac{dE}{dt} = 0$$

$$L\dot{q}\ddot{q} + \frac{1}{C}q\dot{q} = 0$$

$$LC\ddot{q} + q = 0$$

In terms of the flux linkage λ and its derivatives:

$$E = \frac{1}{2L}\lambda^2 + \frac{1}{2}C\dot{\lambda}^2$$

$$\frac{dE}{dt} = \frac{1}{L}\lambda\dot{\lambda} + C\dot{\lambda}\ddot{\lambda} = 0$$

$$LC\ddot{\lambda} + \lambda = 0$$

We could use Lagrange's energy balance equation for conservative systems, which states that:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_n}\right) - \frac{\partial T}{\partial q_n} + \frac{\partial V}{\partial q_n} = 0$$

where

T = total system kinetic energy
V = total system potential energy
n = 1, 2, … refers to the independent coordinates in the system
q_n = generalised coordinate
$\dot{q}_n$ = generalised velocity

$$T = \frac{1}{2}L\dot{q}^2$$

$$V = \frac{1}{2C}q^2$$

$$\frac{\partial T}{\partial \dot{q}} = L\dot{q}$$

$$\frac{d}{dt}\left[\frac{\partial T}{\partial \dot{q}}\right] = L\ddot{q}$$

$$\frac{\partial T}{\partial q} = 0$$

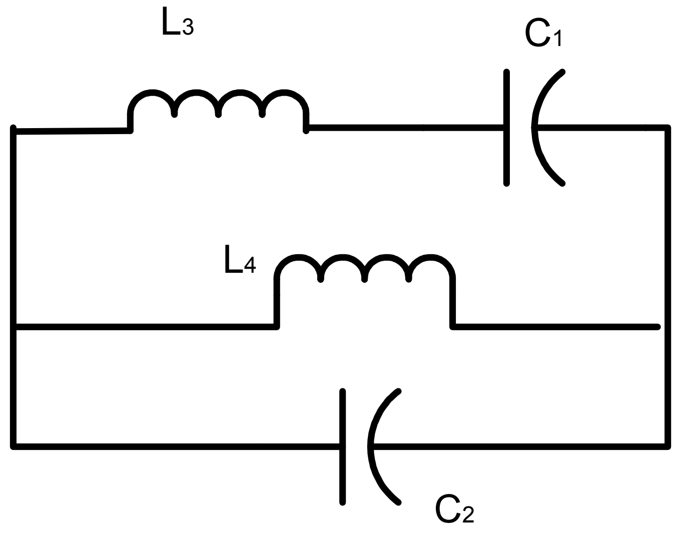
$$\frac{\partial V}{\partial q} = \frac{1}{C}q$$

$$\therefore L\ddot{q} + \frac{1}{C}q = 0$$

$$LC\ddot{q} + q = 0$$

Example 2

Let us consider one more example:



We will consider the flux linkages λ associated with each of the elements as candidates for the selection of independent coordinates.

[The meaning of a flux linkage associated with a capacitor is not quite clear. We will assume that a capacitor is in parallel with an infinite inductor, carrying zero current, but with a flux linkage λ such that its time derivative is equal to the voltage across the capacitor.]

We could select (say) λ_1 and λ_2 as our coordinates (but not λ_2 and λ_4, as they are not independent)

$$T = \frac{1}{2}C_1\dot{\lambda}_1^2 + \frac{1}{2}C_2\dot{\lambda}_2^2$$

$$V = \frac{1}{2L_3}\lambda_3^2 + \frac{1}{2L_4}\lambda_4^2$$

$$= \frac{1}{2L_3}(\lambda_2 - \lambda_1)^2 + \frac{1}{2L_4}\lambda_2^2$$

Evaluating with respect to λ_1 we have:

$$\frac{\partial T}{\partial \dot{\lambda}_1} = C_1\dot{\lambda}_1$$

$$\frac{d}{dt}\left[\frac{\partial T}{\partial \dot{\lambda}_1}\right] = C_1\ddot{\lambda}_1$$

$$\frac{\partial T}{\partial \lambda_1} = 0$$

$$\frac{\partial V}{\partial \lambda_1} = -\frac{1}{L_3}(\lambda_2 - \lambda_1)$$

This yields the state equation:

$$C_1\ddot{\lambda}_1 - \frac{1}{L_3}(\lambda_2 - \lambda_1) = 0$$

Similarly, evaluating with respect to λ$_2$, we get:

$$\frac{\partial T}{\partial \dot{\lambda}_2} = C_2\dot{\lambda}_2$$

$$\frac{d}{dt}\left[\frac{\partial T}{\partial \dot{\lambda}_1}\right] = C_2\ddot{\lambda}_2$$

$$\frac{\partial T}{\partial \lambda_2} = 0$$

$$\frac{\partial V}{\partial \lambda_2} = \frac{1}{L_3}(\lambda_2 - \lambda_1) + \frac{1}{L_4}\lambda_2$$

leading to the state equation:

$$C_2\ddot{\lambda}_2 + \frac{1}{L_3}(\lambda_2 - \lambda_1) + \frac{1}{L_4}\lambda_2 = 0$$

These two equations may now be expressed in standard form by defining a new set of state variables as:

$$x_1 = \lambda_1, \qquad x_2 = \lambda_2,$$
$$x_3 = \dot{\lambda}_1, \qquad x_4 = \dot{\lambda}_2$$

Non-conservative systems

Let us now consider non-conservative systems. The complete Lagrange's equations for such systems are as follows:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_n}\right) - \frac{\partial T}{\partial q_n} + \frac{\partial D}{\partial \dot{q}_n} + \frac{\partial V}{\partial q_n} = Q_n$$

where

T = total system kinetic energy
D = total system dissipation factor, and is defined as one half the rate at which energy is dissipated in the system as heat.
V = total system potential energy
n = 1, 2, … refers to the independent coordinates in the system
Q_n = generalised forcing function relative to coordinate n
q_n = generalised coordinate
$\dot{q}_n$ = generalised velocity

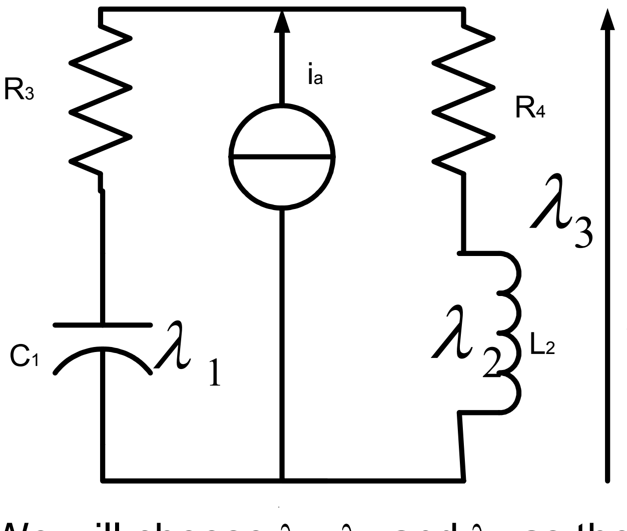
To obtain the forcing function relative to the nth coordinate:

Suppose the system has all its coordinates q_1, q_2, ... , q_n frozen when the system is in an arbitrary configuration. Now, let one coordinate q_i increase by δq_i. Let δw_i be the work done by all external forces in the system during the displacement δq_i. Then,

$$Q_i \,\underline{\Delta}\, \underset{\delta q_i \to 0}{Lim} \frac{\delta w_i}{\delta q_i}$$

Example 3

Let us consider the following example:



We will choose λ_1, λ_2 and λ_3 as the coordinates.

$$T = \frac{1}{2} C_1 \dot{\lambda}_1^2$$

$$V = \frac{1}{2L_2} \lambda_2^2$$

$$D = \frac{1}{2}\left[\frac{(\dot{\lambda}_3 - \dot{\lambda}_1)^2}{R_3} + \frac{(\dot{\lambda} - \dot{\lambda}_2)^2}{R_4} \right]$$

$$Work\, done\, by\, external\, forces =$$

$$i_a v\, dt = i_a \frac{d\lambda_3}{dt} dt = i_a \delta\lambda_3$$

Therefore,

$$\delta w_1 = 0,\ as\ \lambda_1\ is\ held\ const., \quad Q_1 = 0$$

$$\delta w_2 = 0, as\ \lambda_2\ is\ held\ const., \quad Q_2 = 0$$

$$\delta w_3 = i_a \delta\lambda_3 \qquad Q_3 = \frac{i_a \delta\lambda_3}{\delta\lambda_3} = i_a$$

Now let us write down the Lagrange's equations:

$$T = \frac{1}{2} C_1 \dot{\lambda}_1^2$$

$$\frac{d}{dt}\left[\frac{\partial T}{\partial \dot{\lambda}_1}\right] = C_1 \ddot{\lambda}_1,$$

$$\frac{d}{dt}\left[\frac{\partial T}{\partial \dot{\lambda}_2}\right] = 0,$$

$$\frac{d}{dt}\left[\frac{\partial T}{\partial \dot{\lambda}_3}\right] = 0 \qquad .$$

$$\frac{\partial T}{\partial \lambda_1} = 0, \qquad \frac{\partial T}{\partial \lambda_2} = 0, \qquad \frac{\partial T}{\partial \lambda_3} = 0,$$

$$V = \frac{1}{2L_2} \lambda_2^2$$

$$\frac{\partial V}{\partial \lambda_1} = 0, \qquad \frac{\partial V}{\partial \lambda_2} = \frac{\lambda_2}{L_2}, \qquad \frac{\partial V}{\partial \lambda_3} = 0,$$

$$D = \frac{1}{2}\left[\frac{(\dot{\lambda}_3 - \dot{\lambda}_1)^2}{R_3} + \frac{(\dot{\lambda} - \dot{\lambda}_2)^2}{R_4}\right]$$

$$\frac{\partial D}{\partial \dot{\lambda}_1} = -\frac{(\dot{\lambda}_3 - \dot{\lambda}_1)}{R_3},$$

$$\frac{\partial D}{\partial \dot{\lambda}_2} = -\frac{(\dot{\lambda} - \dot{\lambda}_2)}{R_4},$$

$$\frac{\partial D}{\partial \dot{\lambda}_3} = \frac{(\dot{\lambda}_3 - \dot{\lambda}_1)}{R_3} + \frac{(\dot{\lambda}_3 - \dot{\lambda}_2)}{R_4}$$

Substituting in the Lagrange's equation:

$$\frac{d}{dt}\left(\frac{\partial T}{\partial \dot{q}_n}\right)-\frac{\partial T}{\partial q_n}+\frac{\partial D}{\partial \dot{q}_n}+\frac{\partial V}{\partial q_n}=Q_n$$

$$C_1\ddot{\lambda}_1-\frac{(\dot{\lambda}_3-\dot{\lambda}_1)}{R_3}=0 \qquad (1)$$

$$\frac{\lambda_2}{L_2}-\frac{(\dot{\lambda}_3-\dot{\lambda}_2)}{R_4}=0 \qquad (2)$$

$$\frac{(\dot{\lambda}_3-\dot{\lambda}_1)}{R_3}+\frac{(\dot{\lambda}_3-\dot{\lambda}_2)}{R_4}=i_a \qquad (3)$$

From the above, it appears that there are many state variables, as the first derivatives of λ_1, λ_2 and λ_3 and the second derivative of λ_1 appear in the above equations. However, a careful study reveals that most of them can be eliminated:

Substituting from equations (1) and (2) into equation (3) yields:

$$C_1\ddot{\lambda}_1+\frac{\lambda_2}{L_2}=i_a$$

which correspond to only two state equations, as should have been expected for a system with only two energy storage elements.

2.2 Transformations, canonical forms and eigen-values

Introduction

We have seen how the dynamics of a linear, time invariant network may be represented by a set of linear state-space equations, describing the behaviour of the system as well as its output.

Consider the following set of linear equations:

$$
\begin{aligned}
d/dt\,[X] &= AX + BU \\
d/dt\,[Y] &= CX + DU
\end{aligned}
$$

where X represents the state vector and Y is the output. Matrix A is the system matrix. Matrices B, C and D represent the influence of the input on the state, the influence of the state on the output and of the input on the output, respectively.

[We have referred to this as a standard form of representation. A standard form is technically known as a canonical form.]

The state vector X is not unique.

An infinite number of linear transformations of X exist, which are equally valid representations of the system.

Let us consider a simple example.

$$
\begin{aligned}
dx_1/dt &= x_1 + x_2 \\
dx_2/dt &= x_1 - x_2
\end{aligned}
$$

This is a simple second order system, with no excitation.

Now consider a new set of variables z_1 and z_2 such that:

$$
\begin{aligned}
z_1 &= x_1 + 0.4142\, x_2 \\
z_2 &= x_1 - 2.4142\, x_2
\end{aligned}
$$

Then, we have:

$$
\begin{aligned}
x_1 &= 0.8536\, z_1 + 0.1464\, z_2 \\
x_2 &= 0.3536\, z_1 - 0.3536\, z_2
\end{aligned}
$$

Substituting, we get:

$dz_1/dt = 1.414\ z_1$
$dz_2/dt = -1.414\ z_2$

What we have found is that the two systems

$dx_1/dt = x_1 + x_2$
$dx_2/dt = x_1 – x_2$

and

$dz_1/dt = 1.414\ z_1$
$dz_2/dt = -1.414\ z_2$

are equivalent.

We have used a somewhat funny-looking transformation from X to Z, and ended up with an unusual set of equations in Z. It is unusual because the two equations in Z are uncoupled from each other. You will suspect that the transformation we used is not an arbitrary one, but a very special one to yield such a result. On the other hand, it also illustrates that there can be any number of transformations (as long as the two relations are independent of each other) yielding new sets of state variable, which are equally valid descriptions of the system.

The particular form of the equation that we obtained by this unusual transformation is known as a diagonal matrix, and is of great significance in the study of dynamic systems. This form of the system equations was made possible because of a particular property of the system we had chosen, that it has distinct characteristic roots or in other words, distinct eigen-values. Remember that we found that the roots of system functions of RLC networks are simple, so that we will not come across systems with multiple roots in the study of such systems.

We will now look at a more systematic method for the transformation of system matrices with distinct eigen-values to diagonal form. You will find that it is not possible to transform matrices with multiple eigen-values to diagonal form.

The Jordan canonical form is a more inclusive form (that includes the diagonal form for matrices with distinct eigen-values) that allows us to have a common standard or canonical form, similar to any given matrix. Matrices are said to be similar if they have the same eigen-values.

2.2.1 Eigen-values and eigen-vectors

Eigen-value is a hybrid word, made up of the German ‘eigen’ (meaning ‘characteristic’) and the English ‘value’.

As its name implies, its value is characteristic of the system that it represents. We also have other related concepts such as the characteristic equation, characteristic roots etc.

Eigen-values are characteristic of a matrix. In the case of the system equations that we studied, the matrix is the system matrix A.

The eigen-values are given by the solution of the equation:

$$|A - \lambda I| = 0$$

We could of course look at this as the solution to the problem of finding a (non-trivial) solution to:

$$Ax = \lambda x$$

Let us consider the example that we already have:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$|A - \lambda I| = \begin{vmatrix} 1-\lambda & 1 \\ 1 & -1-\lambda \end{vmatrix} = (1-\lambda)(-1-\lambda) - 1$$

$$\therefore \lambda^2 - 2 = 0$$

$$\lambda = \pm\sqrt{2}$$

The eigen-values are +1,414 and –1,414

These correspond to the transformed equation that we obtained, namely,

$$\dot{z}_1 = 1{,}414 z_1$$

$$\dot{z}_2 = -1.414 z_2$$

or, in matrix form:

$$\begin{bmatrix} \dot{z}_1 \\ \dot{z}_2 \end{bmatrix} = \begin{bmatrix} 1.414 & 0 \\ 0 & -1.414 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

We can obtain the eigen-values of a matrix using the function *eig* in MATLAB. The following is a transcript of a MATLAB session, which calculates the eigen-values of the above matrix:

```
A=[1 1; 1 –1]

A=
          1          1
          1         -1

lamda=eig(A)

lamda=
          -1.4142
           1.4142
```

Now let us try some slightly more complex examples:

$$B = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 2 & 2 \\ 1 & 1 & 2 \end{bmatrix}$$

Using MATLAB, let us evaluate their eigen-values:

```
B={1 1 0; 0 1 1; 0 0 1]

B=
          1          1          0
          0          1          1
          0          0          1
eig(B)

ans =

          1
          1
          1
C=[2 2 1; 1 2 2; 1 1 2]
```

```
C=
         2        2        1
         1        2        2
         1        1        2

eig(C)

ans =
         4.6274
         0.6863 + 0.4211i
         0.6863 - 0.4211i
```

These examples illustrate two other phenomena. Matrix B has three coincident eigen-values, while C has one real eigen-value and a pair of complex eigen-values, which are conjugates of each other.

We are now ready to look at another concept, that of eigen-vectors.

Associated with each distinct characteristic value (eigen-value) λ, there is a characteristic vector (eigen-vector), determined up to a scalar multiple. Earlier, we used the relationship

$$Ax = \lambda x$$

to find the eigen-values λ, so that x is non-trivial. We will now find these values of x, corresponding to each eigen-value.

$$(A - \lambda I)\,x = 0$$

for the example considered.

With $\lambda_1 = \surd 2$, the equations are:

$$-0.4142x_1 + x_2 = 0$$

$$x_1 - 2.4142x_2 = 0$$

From either of these, we can obtain

$$x_1 = 2.4142x_2$$

[1/0.4142 is equal to 2.4142, so that both equations yield the same relationship]

If we select $x_2 = 1$, then $x_1 = 2.4142$.

If we normalise this vector, such that the sum of their squares is equal to 1,

$$x_1 = \frac{2.4142}{\sqrt{\left(2.4142^2 + 1^2\right)}} = \frac{2.4142}{2.6131} = 0.9239$$

$$x_2 = \frac{1}{\sqrt{\left(2.4142^2 + 1^2\right)}} = 0.3827$$

If we now start with the other eigen-value
$\lambda_2 = -\sqrt{2}$, we get:

$$2.4142\,x_1 + x_2 = 0$$

$$x_1 + 0.4142\,x_2 = 0$$

Either of these will yield the relationship

$$x_1 = -0.4142\,x_2$$

We can, up to a scalar multiple, assume

$$x_1 = -0.4142$$

$$x_2 = 1$$

After normalisation, we get:

$$x_1 = -0.3827$$

$$x_2 = 0.9239$$

We have now calculated the two eigen-vectors of the matrix A, up to (a) scalar multiple. They are:

$$\begin{bmatrix}0.9239\\0.3827\end{bmatrix},\begin{bmatrix}-0.3827\\0.9239\end{bmatrix}$$

We can use MATLAB to obtain the eigen-vectors of a matrix. Let us illustrate this using the example used:

```
A=[1 1; 1 –1]

A =
         1         1
         1         -1

[V,D] = eig(A)

V =
         0.3827             -0.9239
         -0.9239            -0.3827
```

```
D =
        -1.4142             0
              0        1.4142
```

[Note: When the function *eig* is used with two expected matrix responses as here, the first answer returned is an array of eigen vectors and the second is a diagonal matrix with eigen-values on the diagonal.]

Note that the eigen-vectors computed by MATLAB are different from the ones we obtained, by a factor of (-1). This is because they are determined only up to a scalar multiple, and even normalisation leaves us with this ambiguity.

```
>> B=[1 1 0;0 1 1; 0 0 1]

B =

   1   1   0
   0   1   1
   0   0   1

>> C=[2 2 1; 1 2 2; 1 1 2]

C =

   2   2   1
   1   2   2
   1   1   2

>> [P,Q]=eig(B)

P =

  1.0000          -1.0000           1.0000
     0             0.0000          -0.0000
     0             0                0.0000

Q =

   1   0   0
   0   1   0
   0   0   1

>> [R,S]=eig(C)

R =

  0.6404           0.7792             0.7792
  0.6044          -0.4233 + 0.3106i  -0.4233 - 0.3106i
  0.4738          -0.1769 - 0.2931i  -0.1769 + 0.2931i

S =

  4.6274          0                   0
     0            0.6863 + 0.4211i    0
     0            0                   0.6863 - 0.4211i
```

[Note that this algorithm has not been able to compute the eigen-vectors of the matrix B, as it has multiple eigen-values. An indication of this may be obtained by invoking the function *condeig,* which returns a vector of condition numbers for the evaluation of eigen-vectors. For a well-conditioned matrix, it should have values close to unity.

```
>> condeig(B)

Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 2.465190e-032.
> In D:\matlabR12\toolbox\matlab\matfun\condeig.m at line 30

ans =

  1.0e+031 *

    0.0000
    2.0282
    2.0282
```

]

2.2.2 Diagonal matrices

Let us assume that the matrix A has distinct characteristic roots or eigen-values λ_1, λ_2, λ_3. . . , λ_n. Let us further assume that the eigen-vectors associated with these eigen-values are x^1, x^2, x^3, . . . , x^n. We will further assume that the eigen-vectors have been normalised, so that inner product of each eigen-vector and itself is unity:

$$(x^i, x^i) = \sum_{j=1}^{n} (x_j^i)^2 = 1, for \;\; i = 1, n$$

Consider the matrix T formed by assembling the vectors x^i as columns, as follows:

$$T = \left[x^1 \,|\, x^2 \,|\, x^3 \ldots | \, x^n \right]$$

$$= \begin{bmatrix} x_1^1 & x_1^2 & . & . & x_1^n \\ x_2^1 & x_2^2 & . & . & x_2^n \\ . & . & . & . & . \\ x_n^1 & x_n^2 & . & . & x_n^n \end{bmatrix}$$

T', the transpose of T, would then be the matrix obtained by arranging the eigen-vectors as rows:

$$T' = \begin{bmatrix} x^1 \\ x^2 \\ . \\ . \\ . \\ x^n \end{bmatrix} = \begin{bmatrix} x_1^1 & x_2^1 & . & . & x_n^1 \\ x_1^2 & x_2^2 & . & . & x_n^2 \\ . & . & & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ x_1^n & x_2^n & . & . & x_n^n \end{bmatrix}$$

Since the eigen-vectors are orthogonal, we have:

$$T'T = ((x^i x^j)) = (\delta_{ij})$$

[T is thus an orthogonal matrix.]

As each of the x^i are eigen-vectors, we have:

$$Ax^i = \lambda_i x^i$$

so that:

$$AT = [\lambda_1 x^1 \mid \lambda_2 x^2 \mid . \quad \mid \lambda_n x^n]$$
$$\therefore T'AT = (\lambda_i (x^i x^j)) = (\lambda_i \delta_{ij})$$

$$\therefore T'AT = \begin{bmatrix} \lambda_1 & & & & \\ & \lambda_2 & & 0 & \\ & & \lambda_3 & & \\ . & & & & \\ & 0 & & & \\ & & & & \lambda_n \end{bmatrix}$$

T'AT has eigen-values on the diagonal, and zero everywhere else. It is a diagonal matrix.

Let us define Λ (lamda) = T'AT

Pre-multiplying by T and post-multiplying by T’, we have:

$$T\Lambda T'=TT' A TT' = A$$

For the examples we considered earlier, from the MATLAB simulation, we had:

```
A =

    1    1
    1   -1

T=
         0.3827          -0.9239
         -0.9239         -0.3827
```

Since T is symmetric, T’ = T. and we have:

```
lamda =

  -1.4142   -0.0000
  -0.0000    1.4142
```

This, as expected, is a diagonal matrix with the eigen-values on the diagonal. We can now get back A as $T\Lambda T'$:

```
ans =

   1.0000    1.0000
    1.00      -1.0000
```

2.2.3 The Jordan canonical form

We saw how symmetrical matrices with real eigen-values may be transformed to diagonal form.

The following are two important properties of real symmetric matrices:

- The characteristic roots (eigen-values) of a real symmetric matrix are real.
- The characteristic vectors (eigen-vectors) associated with distinct characteristic roots of a real symmetric matrix are orthogonal.

In practice, we do come across matrices with complex eigen-values. What difference does it make?

Before we proceed, we will need to agree on certain notations.

We have already used the terms *symmetric* matrix and *transpose* of a matrix.

1. A *symmetric matrix* is one where

$$a_{ij} = a_{ji}$$

2. The *transpose* A' of a matrix A is given by:

$$A = (a_{ij})$$
$$A' = (a_{ji})$$

[For a symmetric matrix, A=A']

3. When the elements a_{ij} of a matrix are real, we call such a matrix a *real matrix*.

4. The *inner product* of two vectors x and y is written as (xy), and is an important scalar function of x and y. It is defined as

$$(xy) = \sum_{1}^{n} x_i y_i$$

5. The complex conjugate of a complex variable x is denoted by placing a ¯ over x.

$$x = \alpha + j\beta$$
$$\overline{x} = \alpha - j\beta$$

[For complex vectors, the product $(x, \overline{y})$ is of greater significance than the usual inner product (x,y)]

6. Corresponding to the symmetric real matrices where A = A', the significant form for complex matrices is called *Hermitian matrices*, where:

$$A = \overline{A}' = A^*$$

[Parallel with the orthogonal matrices in the study of symmetric matrices, we have the concept of unitary matrices in the study of Hermitian matrices, where $T^*T = I$]

Let us now consider an example of a non-symmetric matrix with real coefficients:

$$A = \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix}$$

As before, we will look at a MATLAB transcript.

```
» A=[1 -2 ; 2 1]

A =

    1   -2
    2    1

» [T,D]=eig(A)

T =

  -0.7071            -0.7071
       0 + 0.7071i        0 - 0.7071i


D =

   1.0000 + 2.0000i        0
       0             1.0000 - 2.0000i

» TINV= inv(T)

TINV =

  -0.7071                  0 - 0.7071i
  -0.7071                  0 + 0.7071i

» LAMDA=TINV*A*T

LAMDA =

   1.0000 + 2.0000i        0
       0             1.0000 - 2.0000i

» T*LAMDA*TINV

ans =

    1   -2
    2    1

»
```

We first compute the matrix formed by the eigen-vectors (T) and one with the eigen-values as diagonal elements (D). Note that the eigen-values are a complex conjugate pair. We then find the inverse TINV of T. We then note that

$$\Lambda = T^{-1} A T$$

is the diagonal matrix with the eigen-values on the diagonal, the same as D. Finally, we can get back A as:

$$A = T \Lambda T^{-1}$$

[In this particular case, the inverse of T is equal to the transpose of its complex conjugate, but this is not so in general, unless there are only complex roots.]

```
» CTRANS=ctranspose (T)

CTRANS =

  -0.7071                 0 - 0.7071i
  -0.7071                 0 + 0.7071i

» TINV=inv(T)

TINV =

  -0.7071                 0 - 0.7071i
  -0.7071                 0 + 0.7071i
```

Here, the transpose and the inverse are equal.

We will consider one more example.

```
» B=[1 2 3; 2 0 1; 1 2 0]

B =

    1    2    3
    2    0    1
    1    2    0

» [T,D]=eig(B)

T =

  -0.7581           -0.3589 - 0.4523i  -0.3589 + 0.4523i
  -0.4874           -0.2122 + 0.5369i  -0.2122 - 0.5369i
  -0.4332            0.5711 - 0.0847i   0.5711 + 0.0847i


D =

   4.0000                0                 0
       0          -1.5000 + 0.8660i        0
       0                 0           -1.5000 - 0.8660i

» CTRANS=ctranspose(T)

CTRANS =

  -0.7581           -0.4874           -0.4332
  -0.3589 + 0.4523i  -0.2122 - 0.5369i   0.5711 + 0.0847i
  -0.3589 - 0.4523i  -0.2122 + 0.5369i   0.5711 - 0.0847i

» TINV=inv(T)

TINV =
  -0.5957 + 0.0000i  -0.5957 - 0.0000i  -0.5957
  -0.2826 + 0.3820i  -0.1359 - 0.6071i   0.6474 + 0.0145i
  -0.2826 - 0.3820i  -0.1359 + 0.6071i   0.6474 - 0.0145i
```

The transpose and the inverse are not the same.

```
» LAMDA=TINV*B*T

LAMDA =

   4.0000 - 0.0000i  -0.0000 - 0.0000i  -0.0000 + 0.0000i
   0.0000 - 0.0000i  -1.5000 + 0.8660i  -0.0000 - 0.0000i
   0.0000 + 0.0000i  -0.0000 + 0.0000i  -1.5000 - 0.8660i

» T*LAMDA*TINV

ans =

   1.0000 - 0.0000i   2.0000 + 0.0000i   3.0000 - 0.0000i
   2.0000 + 0.0000i        0 + 0.0000i   1.0000 + 0.0000i
   1.0000 - 0.0000i   2.0000 + 0.0000i   0.0000 - 0.0000i
```

Note that transpose of the complex conjugate of T and the inverse of T are quite different. However, we do have a systematic method for transforming a matrix to diagonal form. There is still one assumption we have made, that the eigen-values are distinct.

Let us now look at the case where you get coincident eigen-values.

We will re-examine the example we considered earlier:

$$C = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

```
» C=[1 1 0; 0 1 1; 0 0 1]

C =

    1    1    0
    0    1    1
    0    0    1

» eig(C)

ans =

    1
    1
    1
```

There are three coincident eigen-values, each equal to 1. If you attempt to find the eigen-vectors, you will find that x_1 may take any value, but that the other two components are identically zero. MATLAB also gives the same result.

```
» [T,D]=eig(C)

T =

   1.0000   -1.0000    1.0000
        0    0.0000   -0.0000
        0         0    0.0000


D =

    1    0    0
    0    1    0
    0    0    1
```

T cannot be inverted, as its rank is only 1.

We are now ready to introduce the *Jordan canonical form*.

Definition

Let us denote by $L_k(\lambda)$ a k x k matrix of the form:

$$L_h(\lambda)=\begin{bmatrix} \lambda & 1 & 0 & & & & & 0. \\ 0 & \lambda & 1 & 0 & & . & . & 0 \\ . & . & . & & & & & . \\ . & . & . & . & . & . & . & . \\ & & & & & & \lambda & 1 \\ 0 & 0 & & & & & & \lambda \end{bmatrix}$$

[L_1 would be equal to λ]

It can be shown that there exists a matrix T such that

$$T^{-1}AT=\begin{bmatrix} L_{k_1}(\lambda_1) & & & \\ & L_{k_2}(\lambda_2) & & 0 \\ & & & \\ & 0 & & \\ & & & L_{k_r}(\lambda_r) \end{bmatrix}$$

$$k_1 + k_2 + \ldots + k_r = n$$

$\lambda_1, \lambda_2, \ldots, \lambda_r$ are eigen-values with multiplicity k_1, k_2, . . . k_r.

This representation (that is $T^{-1}AT$) is called the **Jordan Canonical Form.**

The diagonal form we had for the case of distinct eigen-values obviously satisfies this condition.

The matrix

$$C = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

which we picked for study earlier on is already in this form.

We will now consider another example.

Consider

$$A = \begin{bmatrix} 5 & -1 \\ 4 & 1 \end{bmatrix}$$

The eigen-values of A are 3 and 3. We can then write down the Jordan canonical form of the matrix as:

$$J = \begin{bmatrix} 3 & 1 \\ 0 & 3 \end{bmatrix}$$

This does not, however, tell us anything of the transformation T that will yield J from A, other than that:

$$J = T^{-1}AT$$

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:

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

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

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

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:

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

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

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

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

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

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

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

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

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.

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

In the time domain, the response =

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

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:

$$\dot{x}(t) \quad = a_1 + 2a_2 t + 3a_3 t^2 + . \ . \ .$$

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

Equating coefficients of powers of t, we have:

$$a_1 = Aa_0$$
$$a_2 = \frac{1}{2}Aa_1 = \frac{1}{2}A\,Aa_0 = \frac{1}{2}A^2a_0$$
$$a_3 = \frac{1}{3}Aa_2 = \frac{1}{3}A\,\frac{1}{2}A^2a_0 = \frac{1}{2.3}A^3a_0$$

.

.

$$a_r = \frac{1}{r!}A^ra_0$$

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:

$\Delta = 5(25\text{-}4) + 2(\text{-}10\text{-}4)\text{-}2(4+10) = 49$

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

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

Now, writing

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

We have:

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

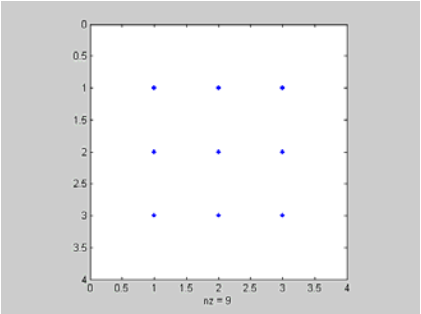
We can use MATLAB to obtain this result using:

```
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
```

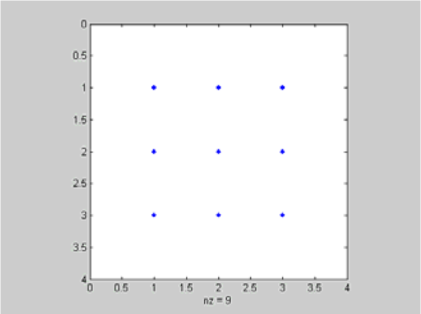
We can use the “spy” instruction to plot the non-zero elements of G:

```
Spy(G)
```



and of G^{-1}:

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

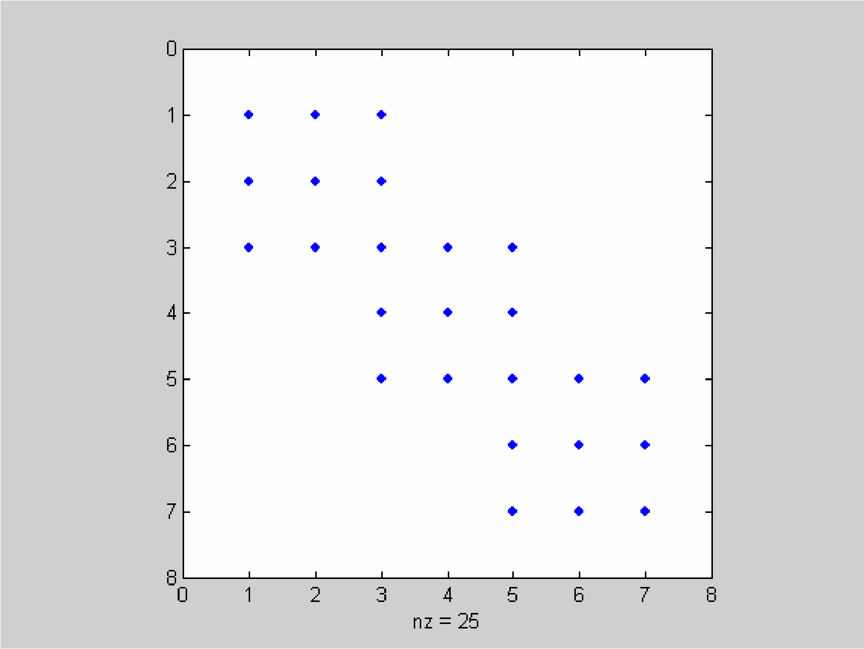


They are both full matrices and nothing (in terms of storage etc.) is gained or lost.

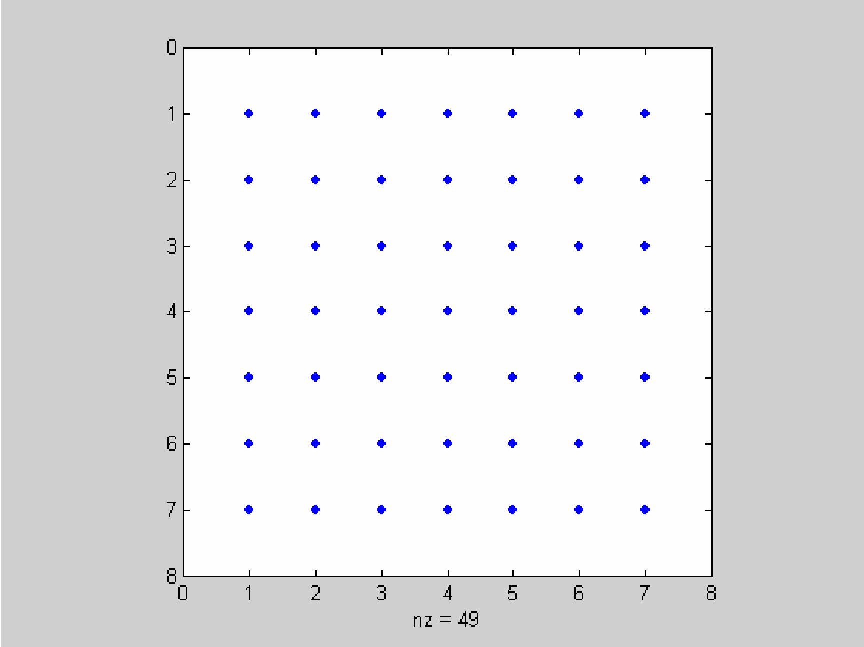
Now lets look at the next example we considered, of three such networks connected in cascade. The non-zero elements of the original matrix and of its inverse are as shown:

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

```
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:
        $b_i = b_i - a_{i1} x b_1$
        For j = 1 to n:
                $a_{ij} = a_{ij} – a_{i1} x a_{1j}$
```

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
```

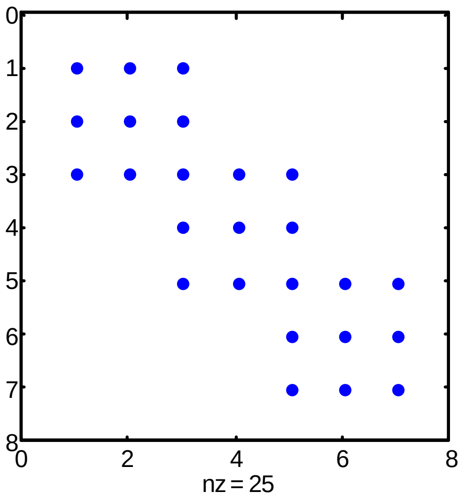
When we need to optimise the use of storage, it is possible to store all the values in one matrix, as it is not necessary to store either the zeros or the 1s. There is also a distinct MATLAM command for this:

```
lu(A)

ans =

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

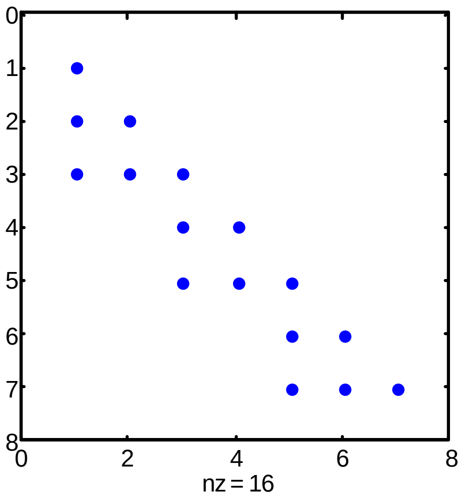
Let us now see the effect of LU factorisation on a sparse matrix.
We will again use the example we considered earlier.

```
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
```

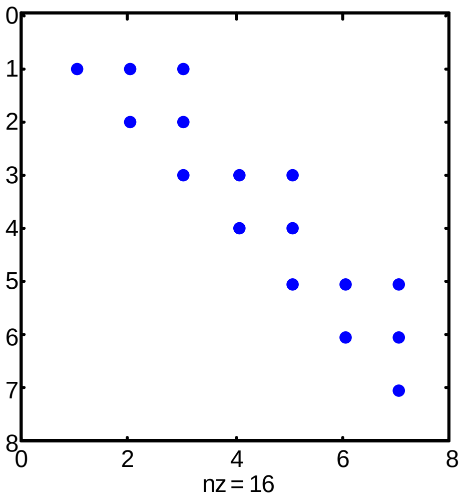
```
spy(A)
```



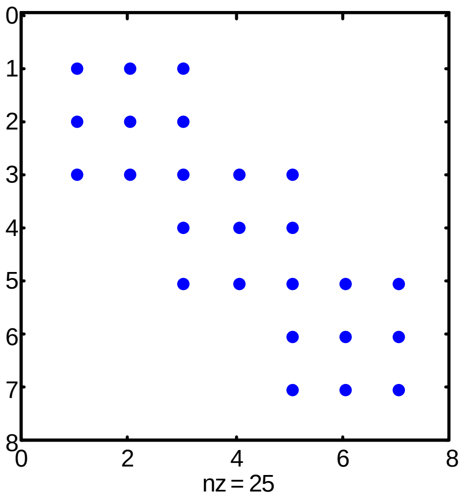
```
» [L,U]=lu(A)

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))

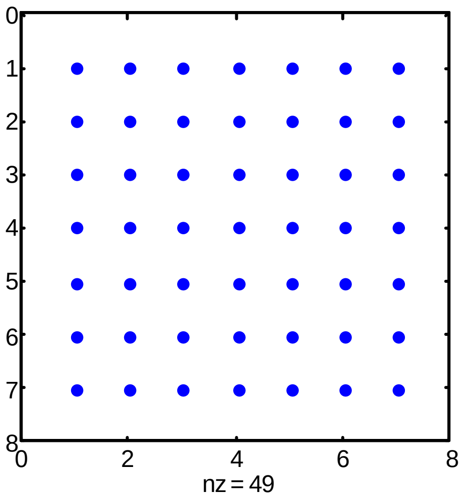


Notice that in this particular case, there has been no increase in storage requirements. This is only if we use one matrix to store both lower and upper triangles, with implied storage of zero and unity values.

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

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

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:

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

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.

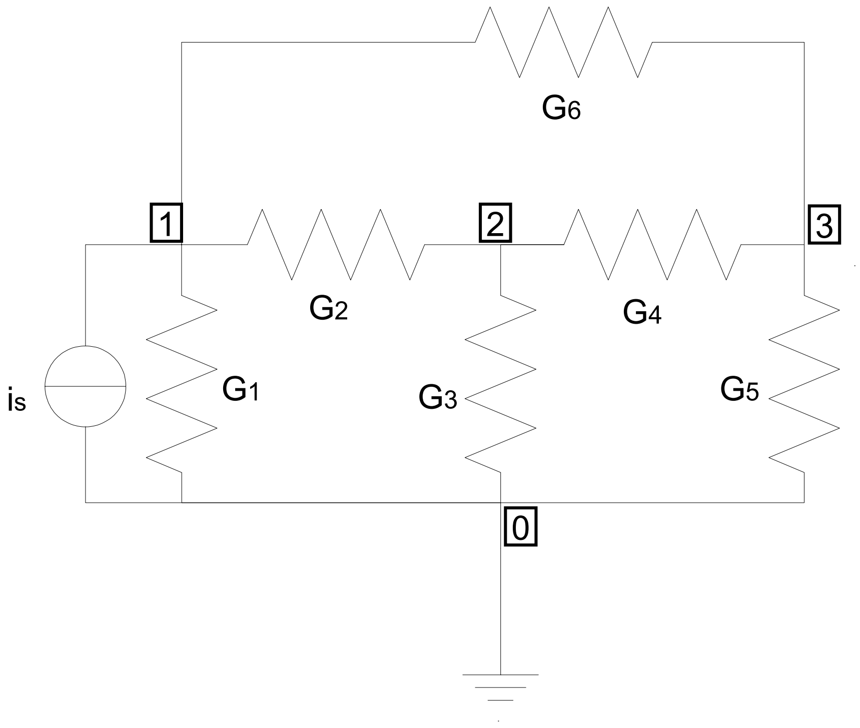
Typically, we encounter PDEs in problems associated with electromagnetic waves where the three space variables and time are all independent variables. PDEs also arise in other branches of engineering, in fluid flow, heat transfer and stree analysis, for example.

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:

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

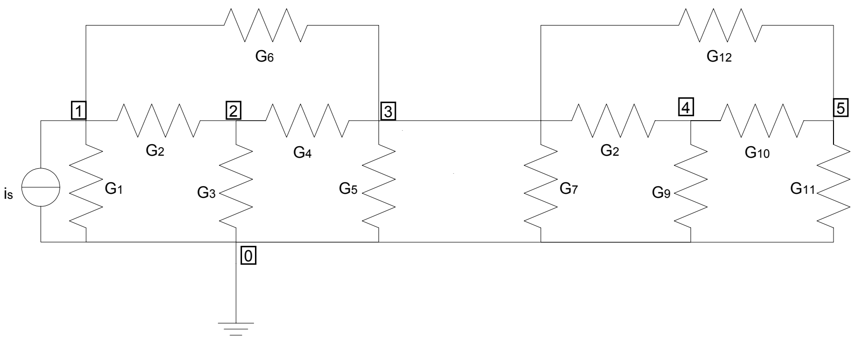
This can be written as:

$$Gv = i$$

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

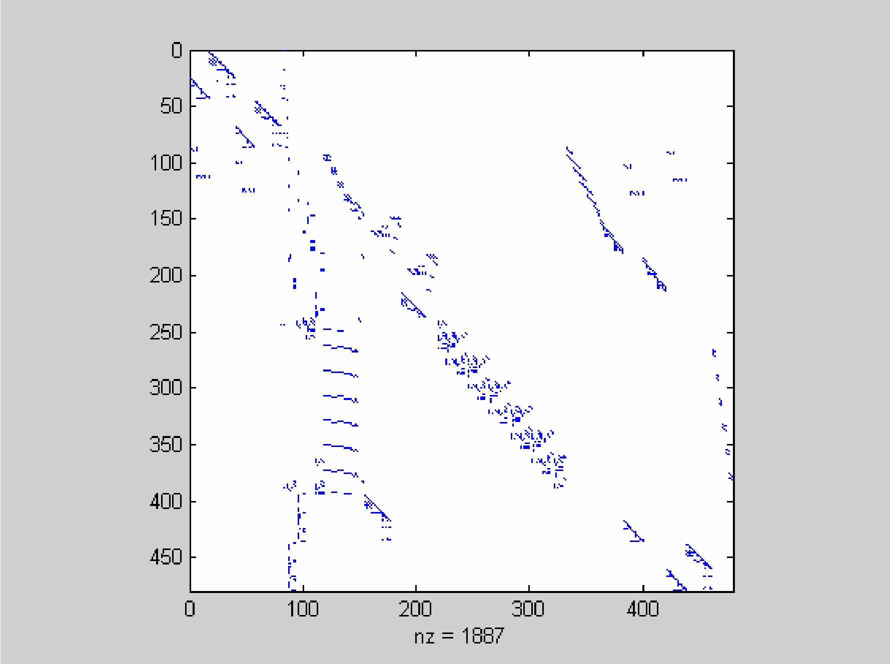
$$\frac{(479*479-1887)}{479*479}*100\,\%$$

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

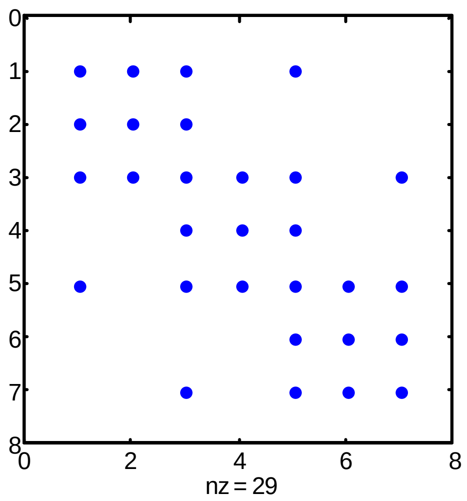
We have already looked at pivoting for reducing round-off errors, when considering Gaussian elimination. In addition to ensuring that the diagonal element be non-zero (a zero diagonal element will lead to a breakdown of the process), it is better that it be comparatively large, as this would reduce computational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

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

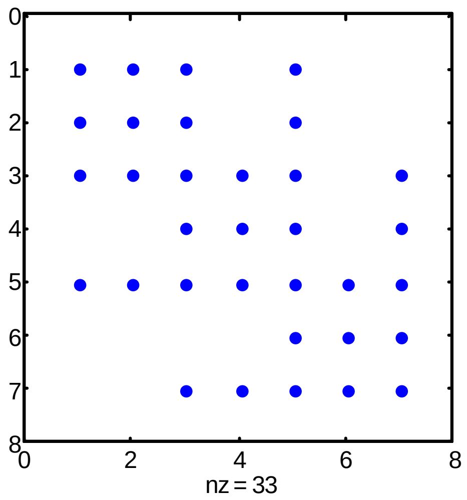
We saw that the processing of the sparse matrix considered in LU factorisation did not result in adding new non-zero elements. This is not always so. We will consider a slightly modified matrix to illustrate this point.

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

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
```

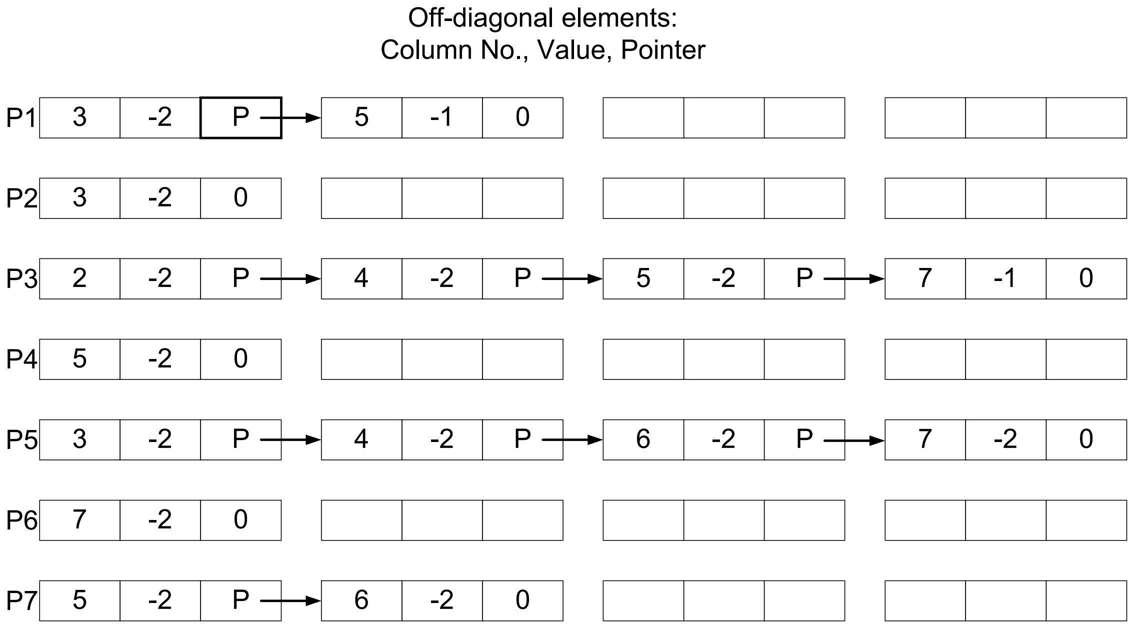
The instruction sparse (A) has converted the storage of the matrix A from its normal form into the sparse matrix representation in MATLAB. As can be seen, this representation uses two integer arrays to indicate the indices of each non-zero element and another real (or complex) array to represent the 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.