5

Hierarchical Modular Modeling of Continuous Systems

Preview

To this point, we have dealt with very simple and small problems. This chapter covers some of the techniques necessary for modeling larger systems. Very often, systems consist of subsystems which may be described in quite different ways. Besides state–space representations and topological descriptions (that we have met previously), subsystems may also be described in the frequency domain in terms of transfer functions, or may simply be given as a static characteristic relating one output variable to one or several input variables. It is therefore important that models can be structured. Modular modeling enables us to encapsulate subsystem descriptions, and treat them as unseparable entities which can be incorporated in a hierarchical fashion within ever more complex system descriptions.

5.1 Modeling Transfer Functions

Let us assume, a system is described by the following transfer function:

$$G(s) = 200 \frac{(s+1)}{(s+10)(s+20)} \tag{5.1}$$

In order to make this system amenable to simulation, we need to convert the specification back from the frequency domain into the time domain. The easiest way to do this is the following.

$$G(s) = \frac{200 + 200s}{200 + 30s + s^2} = \frac{P(s)}{Q(s)} = \frac{Y(s)}{U(s)} \tag{5.2}$$

where $G(s)$ denotes the transfer function, $P(s)$ denotes its numerator polynomial, $Q(s)$ denotes its denominator polynomial, $Y(s)$ denotes the output signal, and $U(s)$ denotes the input signal. We introduce an additional signal $X(s)$

$$G(s) = \frac{Y(s)}{U(s)} = \frac{Y(s)}{X(s)} \cdot \frac{X(s)}{U(s)} \tag{5.3}$$

such that

$$\frac{X(s)}{U(s)} = \frac{1}{Q(s)} \tag{5.4a}$$
$$\frac{Y(s)}{X(s)} = P(s) \tag{5.4b}$$

We now look at eq(5.4a) first. We can rewrite this for our example as:

$$[200 + 30s + s^2]X(s) = U(s) \tag{5.5}$$

which can be transformed back into the time domain as:

$$200x(t) + 30\dot{x}(t) + \ddot{x}(t) = u(t) \tag{5.6}$$

assuming that all initial conditions are zero which is standard practice when operating on transfer functions. We now solve eq(5.6) for its highest derivative:

$$\ddot{x}(t) = -200x(t) - 30\dot{x}(t) + u(t) \tag{5.7}$$

Finally, we introduce the following state variables:

$$\xi_1 = x \tag{5.8a}$$
$$\xi_2 = \dot{x} \tag{5.8b}$$

which leads us to the following state–space model:

$$\dot{\xi}_1 = \xi_2 \tag{5.9a}$$
$$\dot{\xi}_2 = -200\xi_1 - 30\xi_2 + u \tag{5.9b}$$

We now look at eq(5.4b) which can be written for our example as:

$$Y(s) = [200 + 200s]X(s) \tag{5.10}$$

or in the time domain:

$$y(t) = 200x(t) + 200\dot{x}(t) \tag{5.11}$$

and using our state variables:

$$y = 200\xi_1 + 200\xi_2 \tag{5.12}$$

We can rewrite eq(5.9a-b) and eq(5.12) in a matrix form as:

$$\dot{\xi} = \begin{pmatrix} 0 & 1 \\ -200 & -30 \end{pmatrix} \xi + \begin{pmatrix} 0 \\ 1 \end{pmatrix} u \tag{5.13a}$$
$$y = \begin{pmatrix} 200 & 200 \end{pmatrix} \xi \tag{5.13b}$$

In general, if a system is specified through the transfer function:

$$G(s) = \frac{b_0 + b_1 s + b_2 s^2 + \ldots + b_{n-1}s^{n-1}}{a_0 + a_1 s + a_2 s^2 + \ldots + a_{n-1}s^{n-1} + s^n} \tag{5.14}$$

we can immediately convert this to the following state–space description:

$$\dot{\mathbf{x}} = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 & 0 \\ 0 & 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & 0 \\ 0 & 0 & 0 & \ldots & 0 & 1 \\ -a_0 & -a_1 & -a_2 & \ldots & -a_{n-2} & -a_{n-1} \end{pmatrix} \mathbf{x} + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{pmatrix} u \tag{5.15a}$$
$$y = \begin{pmatrix} b_0 & b_1 & b_2 & \ldots & b_{n-2} & b_{n-1} \end{pmatrix} \mathbf{x} \tag{5.15b}$$

In other words, the *state matrix* consists of all zero elements except for a superdiagonal of one elements, and except for the last row in which the negative coefficients of the denominator polynomial are stored. The *input vector* consists of zero elements only except for the last element which is one, and the *output vector* contains the positive coefficients of the numerator polynomial.

This technique will work fine as long as the numerator polynomial is of lower degree than the denominator polynomial. If this is not the case, we need to divide the numerator by the denominator first, and separate in this way the direct *input/output coupling* from the remainder of the system. This procedure will be illustrated by means of another simple example:

$$G(s) = \frac{6s^3 + 32s^2 + 10s + 2}{2s^2 + 8s + 4} \tag{5.16}$$

We always start by normalizing the highest degree coefficient of the denominator polynomial to one, i.e.:

$$G(s) = \frac{3s^3 + 16s^2 + 5s + 1}{s^2 + 4s + 2} \tag{5.17}$$

The division of polynomials works exactly the same way as the division of regular numbers:

$$\begin{array}{rrrrl}
 & (3s^3 + & 16s^2 + & 5s + & 1\,) : (s^2 + 4s + 2) = 3s + 4 \\
- & 3s^3 + & 12s^2 + & 6s & \\
\hline
\backslash & & 4s^2 - & s + & 1 \\
- & & 4s^2 + & 16s + & 8 \\
\hline
 & & \backslash & -17s - & 7
\end{array}$$

i.e., $G(s)$ can also be written as:

$$G(s) = (3s + 4) + \frac{-17s - 7}{s^2 + 4s + 2} \tag{5.18}$$

which can be interpreted as a parallel connection of two subsystems as depicted in Fig.5.1. The transfer function has been split into a polynomial which contains the *direct input/output coupling* of the system, and a remainder transfer function, the numerator of which is now guaranteed to be of lower degree than the denominator polynomial. This contains the so-called *strictly proper* portion of the system.

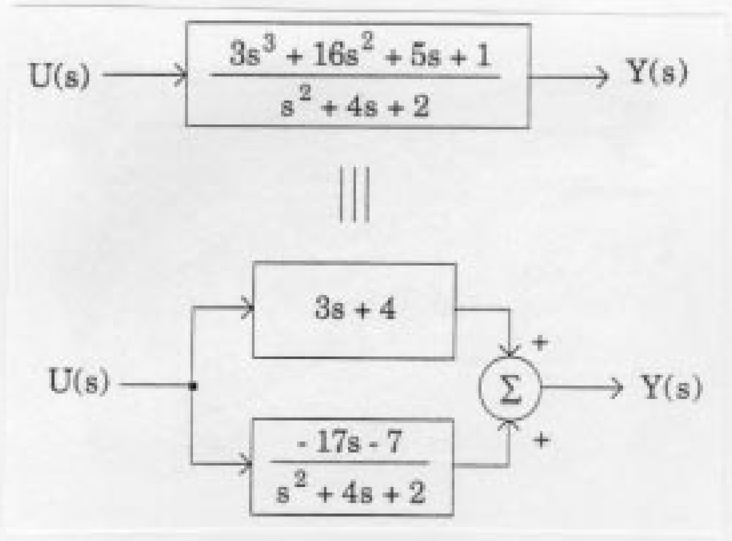
U(s)
$\frac{3s^3 + 16s^2 + 5s + 1}{s^2 + 4s + 2}$
Y(s)
U(s)
3s + 4
+
Σ
Y(s)
$\frac{-17s - 7}{s^2 + 4s + 2}$
+

Figure 5.1. Separation of the direct input/output coupling

In the case of our little example, we end up with the following simulation model:

$$\dot{x}_1 = x_2 \tag{5.19a}$$

$$\dot{x}_2 = -2x_1 + 4x_2 + u \tag{5.19b}$$

$$y = -7x_1 - 17x_2 + 4u + 3\dot{u} \tag{5.19c}$$

As can be seen, a true differentiation of the input signal u was unavoidable in this case. This is always true when the numerator polynomial of a transfer function is of higher degree than the denominator polynomial. As a small consolation: The necessary numerical differentiation is performed as part of the evaluation of output equations, and has thereby been removed from the simulation loop. Numerical errors made in the process of numerical differentiation will not grow by being passed around the integration loop many times. We had met this situation once before in Chapter 3, and at that time, I had mentioned (without a proof) that essential differentiators can, in linear systems, always be moved out of the simulation loop into the output equations. It has now become clear why this is the case, and how this can be accomplished in practice.

5.2 Modeling Static Characteristics

Often, static but non–linear functional relationships exist between input variables and output variables of a subsystem. Often, mathematical equations describing these relationships are not available. Instead, these relationships have been found through experimentation with a real system.

Let us demonstrate this concept by means of our lunar landing module which now should be equipped to land on Earth instead. Of course, this wouldn't work with the rockets designed in the previous model, but let us be forbearing with these lesser details. However, it will be important to modify our mechanical equations to take the air density into consideration. This is proportional to the active surface S, the air density ρ, and the square of the velocity v^2

$$F_{air} = k\ S\ \rho(h)\ v^2 \tag{5.20}$$

The air density is a experimentally determined function of the altitude as depicted in Fig.5.2.

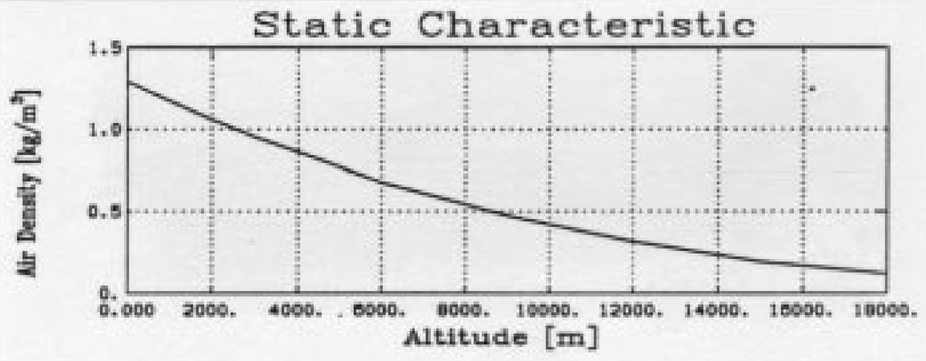

Figure 5.2. Earth's air density as a function of altitude

Most CSSL's provide for mechanisms to describe such functions in a *tabular form*. For instance, DARE–P [5.21] provides for a separate tabular function block in which static characteristics of one and two variables can be coded. The above example could be coded in DARE–P as follows:

```
$ T
 RHO,12
 0.0,     1.293
 300.0,   1.256
 600.0,   1.22
 1200.0, 1.152
 1800.0, 1.082
 3000.0, 0.955
 4500.0, 0.815
 6000.0, 0.676
 9000.0, 0.476
 12000.0,0.319
 15000.0,0.196
 18000.0,0.122
END
```

where the first column denotes the *independent variable*, here altitude expressed in m, and the second column denotes the *dependent variable*, here air density expressed in $kg\ m^{-3}$. The first line specifies the name of the table and the number of recordings collected. This table can be used in the model like a Fortran function, i.e.

$$A = (THRUST - XM * G - XMDOT * V - XK * S * RHO(H) * V * *2)/XM \tag{5.21}$$

ACSL [5.16] uses a different format, but the idea is the same. In ACSL, the above problem would be formulated as follows:

```
table RHO,1,12/ ...
       0.0,    300.0,   600.0, 1200.0,   1800.0, ...
   3000.0,  4500.0,   6000.0,  9000.0, 12000.0, ...
 15000.0, 18000.0,   ...
    1.293,   1.256,    1.22 ,   1.152,     1.082, ...
    0.955,   0.815,    0.676,   0.476,     0.319, ...
    0.196,   0.122/
```

which can also be used like a Fortran function. DESIRE [5.13] offers a similar mechanism, and extensions to several independent variables exist.

5.3 Dynamic Table Load

The tabular function as presented above is not all that useful when we are confronted with huge amounts of data, such as wind tunnel data of an aircraft wing. We certainly don't want to recode (or even re–edit) the data into the format of such tables. For such purpose, it is mandatory that tables can be *loaded dynamically* from a *data base.* Amazingly, this feature was offered already fifteen years ago in the software system CSMP–III [5.9] (*call tvload*), and yet, none of the currently advocated systems offers such a function as a standard feature. However, this function is easy to implement when needed.

5.4 Modular and Hierarchical Modeling

Another requirement that comes immediately to mind is the need to model systems in a *modular* and *hierarchical* manner. It should be possible to create *reusable modules* that can be grouped hierarchically. Modules should be groupable in exactly the same way as real equipment is, i.e., if one wishes to model a cupboard full with electronics, one should be able to model the cupboard as a rack filled with individual instruments, each instrument as a box filled with printed circuit boards, each printed circuit board as a collection of chips, each chip as consisting of a set of transistors, and each transistor through a set of individual discrete elements.

In practice, nobody in his or her right mind would ever attempt to model such a system in all details. Models are always goal driven. The goal dictates the level at which the highest hierarchical layer is placed. Usually, the details of lower hierarchical layers become less and less important to the specified goal, and should be aggregated into atomic units that are not further decomposed. However, it is important that the conceptual mechanism of hierarchical structuring is preserved in the modeling environment. Two hierarchical levels are extremely common in models of practical systems, three to four hierarchical levels can still be found. In Chapter 8, we shall present a model of a solar heated house which contains a five layer hierarchy.

Let us discuss some major mechanisms for hierarchical modeling as they are provided in continuous system modeling environments. By far the most popular mechanism is the *macro facility.*

5.5 The Macro Facility

Most CSSL's allow us to formulate subsystem descriptions as *macros*. Superficially seen, macros look very similar to subprograms in traditional programming languages. This concept will be demonstrated by means of the electro–mechanical system depicted in Fig.5.3.

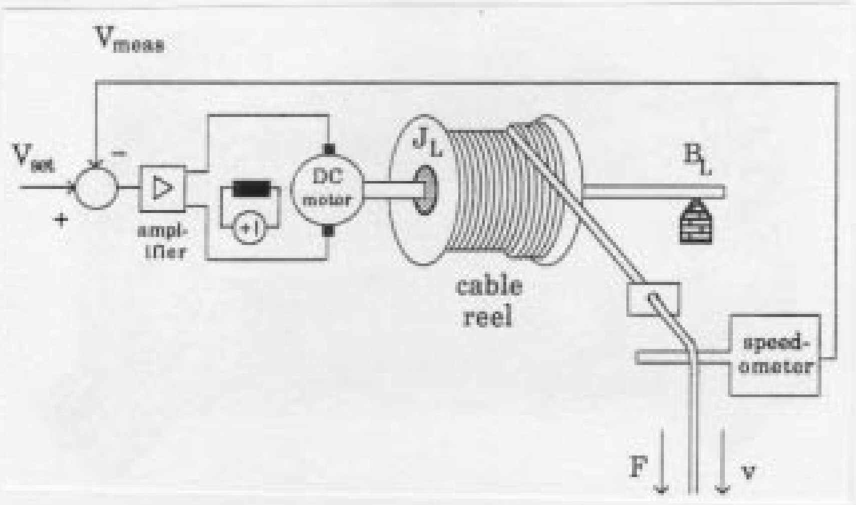

Figure 5.3. Functional diagram of the cable reel system

A new light weight fiber optics deep sea communication cable is to be laid through the British Channel between Calais in France and Dover in the United Kingdom. The cable comes on a huge reel which is placed on a ship. The ship moves slowly from one coast to the other by constantly leaving cable behind. A large DC–motor unrolls the cable from the reel. A speedometer detects the speed of the cable as it comes off the reel. A simple proportional and integral (PI) controller is used to keep the cable speed v at its preset value V_{set}.

In modeling this system, we must first realize that we cannot use the block diagram of the DC–motor as it was developed in the last chapter. The reason is that the inertia J_L of the cable reel changes with time. Consequently, we must use the modified version of Newton's law by operating on the twist $\mathcal{T}$ of the motor. Fig.5.4 shows a modified block diagram of the DC–motor that will work correctly for our application.

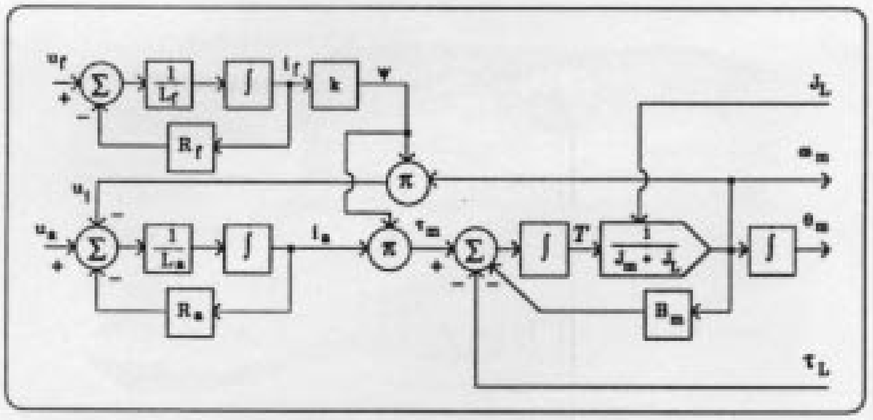

Figure 5.4. Modified block diagram of a DC–motor

The modified version of Newton's law manifests itself through a swapping of the inertial box and one of the integrator boxes in the mechanical subsystem. Notice also that it was necessary to introduce a fourth input variable, namely the inertia of the load, J_L. It wouldn't have been a good idea to combine this term with the torque load (by treating it as an fictitious "inertial torque") since, under those circumstances, an algebraic loop would have resulted.

Let us now analyze the dynamics of the cable reel itself. The length of one winding of the cable is obviously:

$$\ell_w = 2\pi R \tag{5.22}$$

where R is the current diameter of the cable reel. With the width of the cable reel being W, and the diameter of the cable being D, the length of one cable layer can obviously be computed to:

$$\ell_l = \ell_w \frac{W}{D} = \frac{2\pi R\,W}{D} \tag{5.23}$$

The velocity of the cable can be computed to be:

$$v = R\frac{d\theta_m}{dt} \tag{5.24}$$

and since the velocity was assumed to be approximately constant, we can approximate the time to unroll one layer of cable as:

$$t_l = \frac{\ell_l}{v} = \frac{2\pi W}{D\omega_m} \tag{5.25}$$

During this time, the radius of the cable reel is reduced by one cable diameter:

$$\Delta R = -D \tag{5.26}$$

and therefore, we can find a differential equation describing the change of the cable reel radius as follows:

$$\frac{dR}{dt} \approx \frac{\Delta R}{t_l} = -\frac{D^2}{2\pi W}\omega_m \tag{5.27}$$

The inertia of the cable reel is a function of the radius R. It can be computed by the following formula:

$$J_L = 0.5\pi W\rho(R^4 - R^4_{empty}) + J_0 \tag{5.28}$$

where ρ denotes the density of the cable material, and J_0 denotes the inertia of the empty cable reel. The torque load τ_L consists of the friction torque, and the torque produced by the force F which is a result of the weight of the already laid cable. However, this force *supports* the DC–motor rather than impeding it. Therefore, this term is entered with opposite sign

$$\tau_L = B_L\omega_m - F\ R \tag{5.29}$$

Fig.5.5 shows a block diagram of the cable reel dynamics.

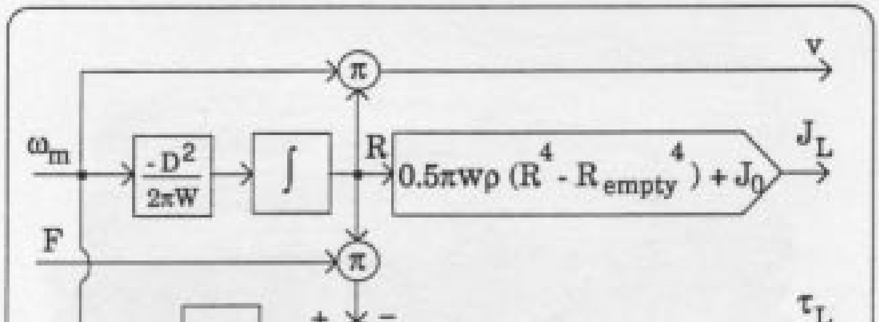

Figure 5.5. Block diagram of cable reel dynamics

Finally, we need to look into the internal dynamics of the speedometer. These were specified by the manufacturer in the frequency domain as:

$$G(s) = \frac{3}{s+3} \tag{5.30}$$

which can immediately be transformed into the state-space representation:

$$\dot{\xi} = -3\xi + u \tag{5.31a}$$
$$y = 3\xi \tag{5.31b}$$

We could now go ahead and simply concatenate all these equations to a monolithic program. However, in a sufficiently large model, this approach is certainly error prone. It seems desirable to be able to formulate the program through modules which represent the structural components of the system as depicted in Fig.5.6.

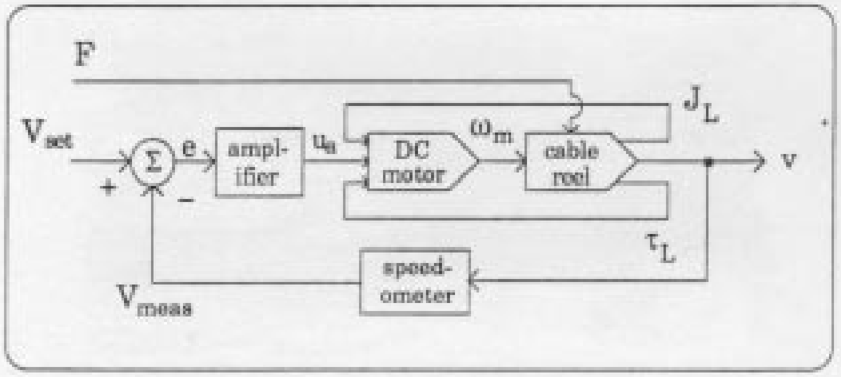

Figure 5.6. Block diagram of the overall cable reel system

This can be achieved by using the *macro facility.* Let me demonstrate this facility by means of the ACSL [5.16] language (DARE–P [5.21] does not offer a macro facility, and while DESIRE [5.13] provides for such a facility, it is much less powerful than the one offered in ACSL). The following macro describes the speedometer:

```
MACRO TACHO(out, in, x0)
  MACRO redefine x, xdot
  MACRO standval x0 = 0.0
    xdot = -3 * x + in
    x    = INTEG(xdot, x0)
    out  = 3 * x
MACRO END
```

This macro can then be utilized in the ACSL program by use of the statement:

$$TACHO(vmeas = v)$$

the *STANDVAL* directive allows us to assign default values to parameters that can then be omitted in the macro call. The *REDEFINE* directive is necessary due to the SAL rule. If the model would contain several speedometers, several "$xdot = \ldots$" statements would otherwise appear in the resulting program after the macro call has been replaced by the macro definition, and the equation sorter would complain. The *REDEFINE* directive instructs the macro handler to replace each occurrence of the names x and $xdot$ during macro replacement by a unique identifier (in ACSL, a Z followed by 5 digits). In other words, if it is intended to use a macro several times within a program, all local variables of the macro must be *declared* in a *REDEFINE* statement.

Let us now write a macro to describe the cable reel dynamics:

```
MACRO CABREL(v, tauL, JL, R, omega, F, D, W, rho, ...
             Rfull, Rempty, J0, BL)
  MACRO redefine Rdot
  constant pi = 3.14159
    Rdot = -((D * D)/(2.0 * pi * W)) * omega
    R    = INTEG(Rdot, Rfull)
    v    = R * omega
    JL   = 0.5 * pi * W * rho * (R ** 4 - Rempty ** 4) + J0
    tauL = BL * omega - F * R
MACRO END
```

From this macro, we can learn several new lessons. First, we may notice the long parameter list. While it would be perfectly acceptable *not* to list all the constants among the input parameters of the macro, this would prevent us from simulating several cable reels at once with different values for these parameters. Unfortunately, while the macro facility provides us with a mechanism to hierarchically decompose *program structures*, it does not allow us to hierarchically decompose the *data structures* along with the program structures. We shall discuss later in this chapter another mechanism that will provide us with such a feature. Clearly, when using the macro facility, all parameter values must be passed on to higher and higher hierarchical levels until the calling sequences become totally unmanageable. This is a serious drawback of the macro concept. Also, we have learned before that it helps the efficiency of the program execution if all *constant computations*, such as $-\frac{D^2}{2\pi W}$ and R^4_{empty}, are moved out of the *DERIVATIVE* section into the *INITIAL* section of the program. We cannot do so in this case without destroying the integrity of the macro. Unfortunately, few CSSL macro handlers have been devised to contain an *INITIAL* section (although this would be quite easy to implement). One simulation language that offers such a facility is SYSMOD [5.19].

Now, let us look at the macro describing the dynamics of the DC–motor.

```
MACRO DCMOT(theta, omega, ua, uf, tauL, JL, ...
  Ra, La, Rf, Lf, k, Jm, Bm, flag, if0, ia0, T0, th0)
  MACRO redefine ia, iadot, if, ifdot, ui, psi
  MACRO redefine taum, Twist, Tdot
  MACRO standval if0 = 0.0, ia0 = 0.0, T0 = 0.0
  MACRO standval th0 = 0.0
  MACRO if (flag = IND) labind
    if     = uf/Rf
    ia     = (ua - ui)/Ra
  MACRO goto goon
  MACRO labind..continue
    ifdot  = (uf - Rf * if)/Lf
    if     = INTEG(ifdot, if0)
    iadot  = (ua - ui - Ra * ia)/La
    ia     = INTEG(iadot, ia0)
  MACRO goon..continue
    psi    = k * if
    taum   = psi * ia
    ui     = psi * omega
    Tdot   = taum - tauL - Bm * omega
    Twist  = INTEG(Tdot, T0)
    omega  = Twist/(Jm + JL)
    theta  = INTEG(omega, th0)
MACRO END
```

Also this macro exhibits a number of additional features. Let us look once more at the armature equation:

$$u_a = u_i + R_a\ i_a + L_a \frac{di_a}{dt} \tag{5.32}$$

Usually, eq(5.32) will be implemented in the form of a differential equation in state–space form, i.e.:

$$\frac{di_a}{dt} = \frac{u_a - u_i - R_a\ i_a}{L_a} \tag{5.33}$$

This is done since we always wish to transform continuous models into a state–space form such that all differential equations can be numerically integrated instead of being numerically differentiated. However, the electrical time constants of the DC–motor are often so much smaller than the mechanical time constant that the effect of the armature inductance on the overall system behavior can be neglected. In this case, we cannot operate on eq(5.33) since, if we set $L_a = 0.0$, this results in a division by zero. Instead, we must return to eq(5.32), delete the term in L_a from the equation, and rewrite it as:

$$i_a = \frac{u_a - u_i}{R_a} \tag{5.34}$$

Notice that this example confronts us with two different versions of DC–motor models. Instead of creating two separate macros for these two cases, it was decided to code them as two variants within the same macro. The constant parameter *flag* can assume either the value *IND* or *NOIND* in the macro call. Depending on the setting of this compile–time parameter, the macro replacer will generate code either in the form of eq(5.33) or in the form of eq(5.34).

We are now ready to code the entire cable reel simulation program.

```
PROGRAM Cable Reel Dynamics
  INITIAL
    MACRO TACHO(out, in, x0)
      ...
    MACRO END
    MACRO CABREL(...)
      ...
    MACRO END
    MACRO DCMOT(...)
      ...
    MACRO END
    constant ...
      Rfull = 1.2, Rempty = 0.6, W = 1.5, D = 0.0127, ...
      rho = 1350.0, J0 = 150.0, Jm = 5.0, Bm = 0.2, ...
      BL = 6.5, Ra = 0.25, La = 0.5E - 3, Rf = 1.0, ...
      Lf = 0.002, kmot = 1.5, kprop = 6.0, kint = 0.2, ...
      vset = 15.0, kship = 10.0, F0 = 100.0, tmx = 3600.0
    cinterval cint = 1.0
    algorithm ialg = 2
    nsteps nstp = 1000
    uf = 25.0
  END $ "of INITIAL"
  DYNAMIC
    DERIVATIVE
      err    = vset - vmeas
      ua     = kprop * err + kint * INTEG(err,0.0)
      DCMOT(theta, omega = ua, uf, tauL, JL, ...
              Ra, La, Rf, Lf, kmot, Jm, Bm, "NOIND")
      CABREL(v, tauL, JL, R = omega, F, ...
              D, W, rho, Rfull, Rempty, J0, BL)
      F      = AMAX1(kship * (vset - v), F0)
      vmeas= TACHO(v)
    END $ "of DERIVATIVE"
    termt (t.ge.tmx .or. R.lt.Rempty)
  END $ "of DYNAMIC"
END $ "of PROGRAM"
```

As the *TACHO* call demonstrates, it is allowable to call macros that produce only one single output as a *function* instead of as a *procedure*. The *ALGORITHM* and *NSTEPS* instructions were necessary to keep the numerical integration happy. The meaning of these instructions will be discussed later. It made little sense to model the electrical time constant of the DC–motor. This time constant is so much smaller than the mechanical time constants that the results would look just the same if the inductances were included, but the simulation would execute much more slowly since the step size of the integration algorithm must adapt itself to the fastest time constant in the system.

The results of this simulation are shown in Fig.5.7.

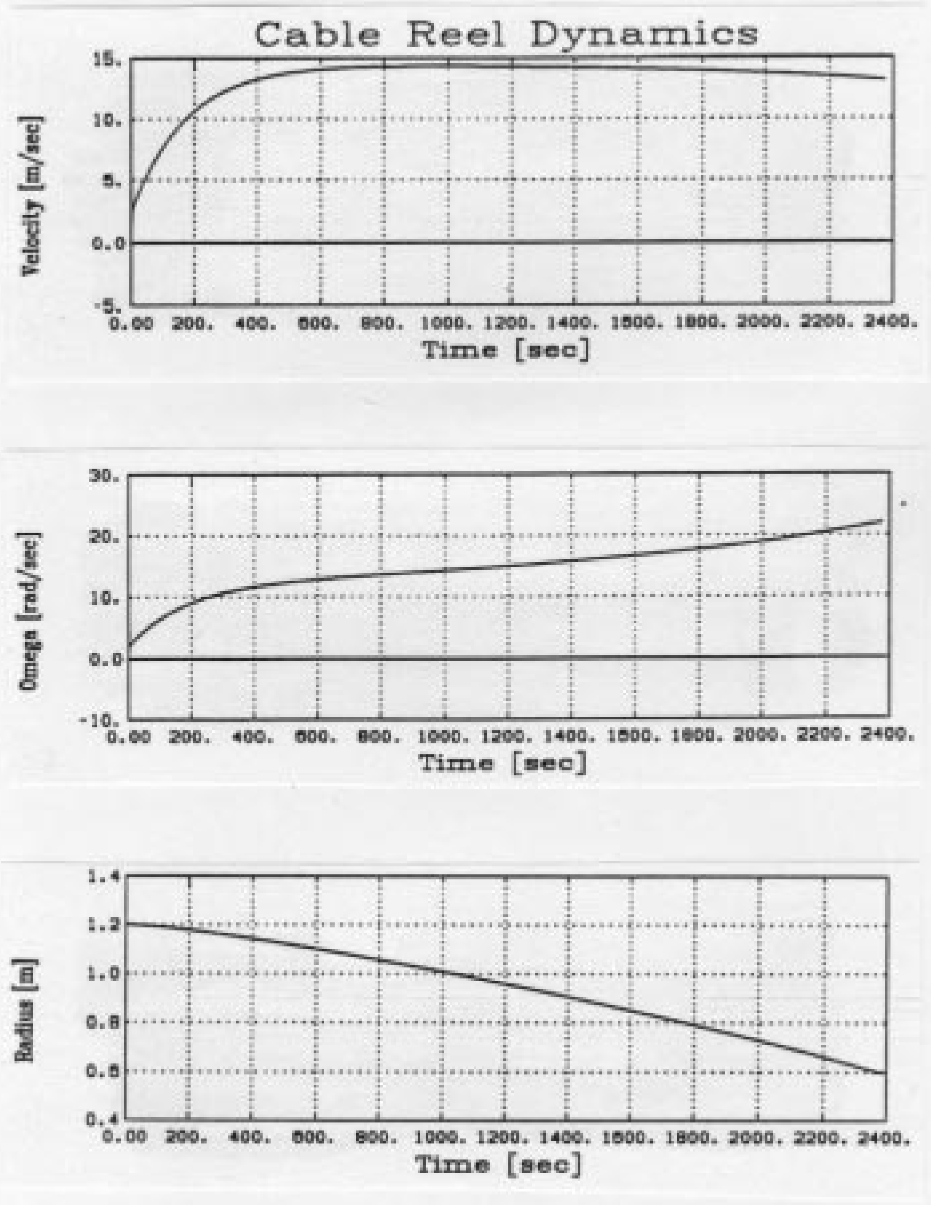

Figure 5.7. Simulation results of cable reel dynamics

As can be seen, it took almost 10 *min* to accelerate the cable reel to its steady–state speed. Obviously, the DC–motor is a little weak for this application, and it might have made sense to replace it by a more sturdy hydraulic motor. However, the control worked just fine. The motor accelerates constantly in order to keep the speed of the unrolling cable at a constant value.

Let us now look at the equations that the macro handler generates during the macro expansion. For illustration, the inductances were included this time.

$$
\begin{array}{lll}
\textbf{main:} & err & = vset - vmeas \\
 & ua & = kprop * err + kint * Z09987 \\
 & Z09987 & = INTEG(err, 0.0) \\
\textbf{dcmot:} & Z09996 & = (uf - Rf * Z09997)/Lf \\
 & Z09997 & = INTEG(Z09996, 0.0) \\
 & Z09998 & = (ua - Z09995 - Ra * Z09999)/La \\
 & Z09999 & = INTEG(Z09998, 0.0) \\
 & Z09994 & = kmot * Z09997 \\
 & Z09993 & = Z09994 * Z09999 \\
 & Z09995 & = Z09994 * omega \\
 & Z09991 & = Z09993 - tauL - Bm * omega \\
 & Z09992 & = INTEG(Z09991, 0.0) \\
 & omega & = Z09992/(Jm + JL) \\
 & theta & = INTEG(omega, 0.0) \\
\textbf{cabrel:} & Z09990 & = -((D * D)/(2.0 * 3.14159 * W)) * omega \\
 & R & = INTEG(Z09990, Rfull) \\
 & v & = R * omega \\
 & JL & = 0.5 * 3.14159 * W * rho * (R * *4 - Rempty * *4) + J0 \\
 & tauL & = BL * omega - F * R \\
\textbf{main:} & F & = AMAX1(kship * (vset - v), F0) \\
\textbf{tacho:} & Z09988 & = -3 * Z09989 + v \\
 & Z09989 & = INTEG(Z09988, 0.0) \\
 & vmeas & = 3 * Z09989
\end{array}
$$

We recognize that, while the *REDEFINE* declaration is necessary for the proper functioning of the macro mechanism, it does not exactly contribute to the readability of the generated code.

Let us go one step further, and *sort* the above equations into an executable sequence. The result is as follows.

$$\begin{array}{lll}
\textbf{demot:} & Z09996 & = (uf - Rf * Z09997)/Lf \\
 & Z09994 & = kmot * Z09997 \\
 & Z09993 & = Z09994 * Z09999 \\
\textbf{cabrel:} & JL & = 0.5 * 3.14159 * W * rho * (R * *4 - Rempty * *4) + J0 \\
\textbf{tacho:} & vmeas & = 3 * Z09989 \\
\textbf{main:} & err & = vset - vmeas \\
 & ua & = kprop * err + kint * Z09987 \\
\textbf{demot:} & omega & = Z09992/(Jm + JL) \\
\textbf{cabrel:} & Z09990 & = -((D * D)/(2.0 * 3.14159 * W)) * omega \\
 & v & = R * omega \\
 & tauL & = BL * omega - F * R \\
\textbf{main:} & F & = AMAX1(kship * (vset - v), F0) \\
\textbf{tacho:} & Z09988 & = -3 * Z09989 + v \\
\textbf{demot:} & Z09995 & = Z09994 * omega \\
 & Z09991 & = Z09993 - tauL - Bm * omega \\
 & Z09998 & = (ua - Z09995 - Ra * Z09999)/La \\
\textbf{main:} & Z09987 & = INTEG(err, 0.0) \\
\textbf{demot:} & Z09997 & = INTEG(Z09996, 0.0) \\
 & Z09999 & = INTEG(Z09998, 0.0) \\
 & Z09992 & = INTEG(Z09991, 0.0) \\
 & theta & = INTEG(omega, 0.0) \\
\textbf{cabrel:} & R & = INTEG(Z09990, Rfull) \\
\textbf{tacho:} & Z09989 & = INTEG(Z09988, 0.0)
\end{array}$$

A number of different algorithms exist that the *equation sorter* can use. One algorithm is the following. We start by assuming all outputs of *memory functions* (i.e., the outputs of integrators) to be known. We then skim through the equations, and try to find one which does not define a memory function, and which has only known variables to the right of the equal sign. If we found one, we write it to the output file, and add the defined variable to the set of known variables. When we reach the end of the input file, we check whether any equations were written to the output file during this pass. If not, we obviously have one or several *algebraic loops*, and another algorithm is being activated to detect algebraic loops among the remaining equations. On the other hand, if we have written one or several equations to the output file, we check whether the set of remaining equations is now empty or contains only memory function definitions, otherwise we go back, and start all over again with the remaining equations. We continue until the set of unsorted equations contains only memory function definitions which can then be added to the output file, or until no more equations can be written to the output file due to the presence of algebraic loops.

We learn from the above example that, during the sorting process, equations from the various macros are being completely interspersed. As a consequence, the macro replacement must be performed *before* the equation sorter can be activated, and consequently, macros cannot be separately compiled, but must always be stored as source code. In this respect, macros are totally different from the subprograms of general purpose programming languages.

This last observation explains why DESIRE's [5.13] macro facility is not very powerful. Since DESIRE does not provide for an equation sorter, macros can only be coded for subsystems that do not require sorting of the equations after the macro replacement has taken place. This makes DESIRE's built in macro facility virtually worthless. However, in the next section, we shall introduce an alternative utility which can replace the macro handler altogether, and which will work fine also in connection with DESIRE.

Let us now look at another example which will unravel some more of the shortcomings of the macro facility as it is offered in today's CSSL's. One of the requests that we may have when using a DC–motor is that the angular velocity of the motor ω_m is insensitive to changes in the torque load T_L. Unfortunately, this is not the case in the standard armature control configuration. Let us look once more at the equations describing the dynamics of the DC–motor:

$$\frac{di_a}{dt} = \frac{u_a - u_i - R_a\ i_a}{L_a} \tag{5.35a}$$

$$\tau_m = \psi\ i_a \tag{5.35b}$$

$$u_i = \psi\ \omega_m \tag{5.35c}$$

$$\frac{d\omega_m}{dt} = \frac{\tau_m - \tau_L}{J_m + J_L} \tag{5.35d}$$

In *steady–state*, all derivatives are zero, i.e., eq(5.35a-d) can be rewritten as:

$$0.0 = u_a - u_i - R_a\ i_a \tag{5.36a}$$

$$\tau_m = \psi\ i_a \tag{5.36b}$$

$$u_i = \psi\ \omega_m \tag{5.36c}$$

$$0.0 = \tau_m - \tau_L \tag{5.36d}$$

which can be reduced to:

$$\omega_m = \frac{1}{\psi} u_a - \frac{R_a}{\psi^2} \tau_L \tag{5.37}$$

Eq(5.37) can be graphically depicted as follows (Fig.5.8):

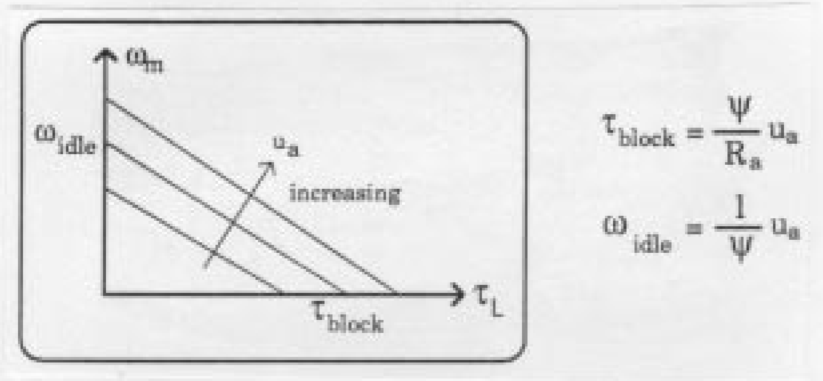

Figure 5.8. Angular velocity *vs* torque load in DC–motor

In order to keep the angular velocity constant under the influence of a changing load, it is necessary to change the armature voltage along with the load. However, the load is usually considered to be an unpredictable *disturbance* of the system. For this reason, armature control may not be the best of all choices. However, if we are able to feed the armature circuit with a constant current source, and apply field control, the dependency of the angular velocity from the torque load vanishes. Unfortunately, constant current sources for high power applications aren't so easy to come by. A more practical solution to the problem is to use a Ward–Leonard group as depicted in Fig.5.9.

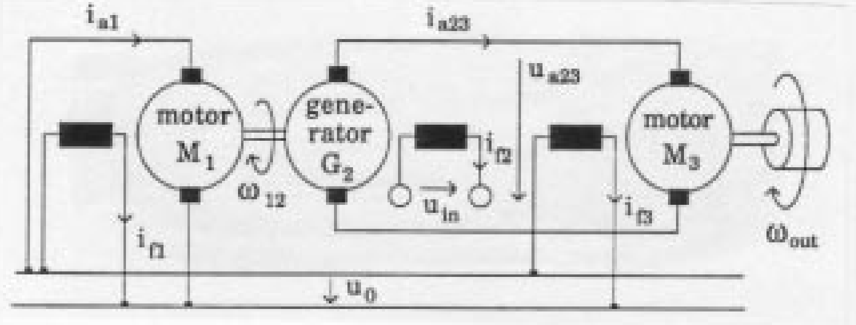

Figure 5.9. Ward–Leonard group

The DC-motor M_1 is driven with constant armature and constant field, and thereby produces an angular velocity which is used to drive the generator G_2. The generator is a machine of exactly the same type as the motor. The electro-mechanical coupling works both ways. We can either generate a rotation by having currents flow through both the field and the armature circuits (motor), or we can induce a voltage in the armature circuit by rotating the machine externally if we feed current through the field circuit at the same time (generator). Fig.5.10 depicts a functional diagram of the DC-generator.

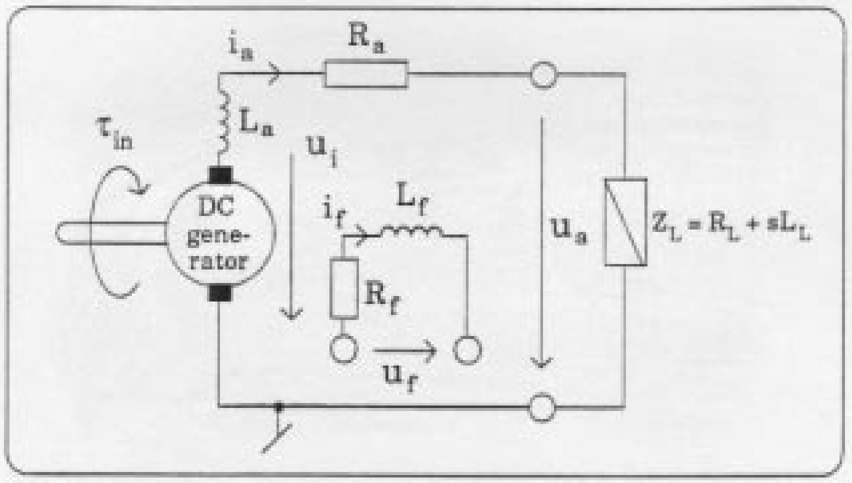

Figure 5.10. Functional diagram of a DC-generator

When the machinery is operated in its *generator* mode, the main input to the system is the driving torque. It causes the motor to rotate. Once an angular velocity ω has been built up, it induces a voltage u_i on the electrical side which causes a current i_a to flow through the armature coil. The armature current i_a causes a mechanical torque τ_L to be built up back on the mechanical side which opposes the driving torque. The armature current i_a is also responsible for building up an armature voltage u_a across the two armature terminals. The armature voltage u_a is subtracted from the induced voltage u_i thereby weakening the armature current i_a. This process continues until an equilibrium is reached. The load is now electrical, symbolized in our model by a resistive load R_L and an inductive load L_L which, in themselves, are not part of the DC–generator, and are therefore additional inputs to the DC–generator model. Fig.5.11 depicts the block diagram of the DC–generator.

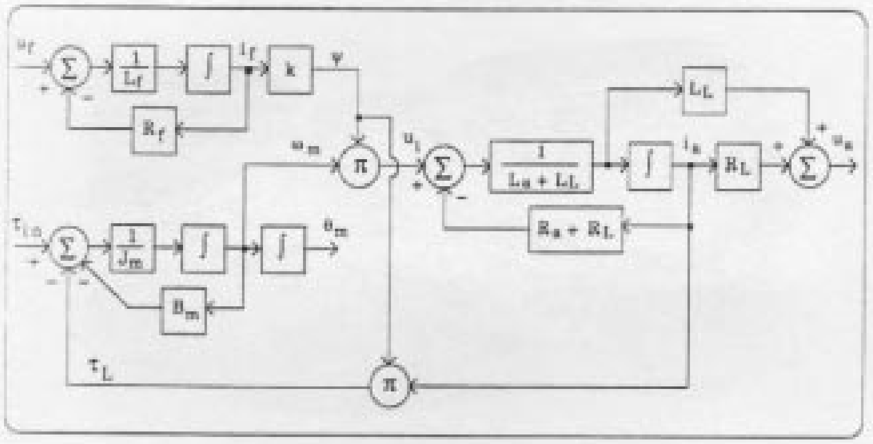

Figure 5.11. Block diagram of a DC–generator

The following ACSL macro can be used to code the DC–generator equations.

```
MACRO DCGEN(theta, omega, ua, ia, tauin, uf, RL, LL, ...
  Ra, La, Rf, Lf, k, Jm, Bm, if0, ia0, om0, th0)
  MACRO redefine iadot, if, ifdot, ui, psi
  MACRO redefine tauL, omdot
  MACRO standval if0 = 0.0, ia0 = 0.0, om0 = 0.0
  MACRO standval th0 = 0.0
  ifdot  = (uf - Rf * if)/Lf
  if     = INTEG(ifdot, if0)
  iadot  = (ui - (Ra + RL) * ia)/(La + LL)
  ia     = INTEG(iadot, ia0)
  ua     = RL * ia + LL * iadot
  psi    = k * if
  tauL   = psi * ia
  ui     = psi * omega
  Tdot   = tauin - tauL - Bm * omega
  Twist  = INTEG(Tdot, T0)
  omega  = Twist/Jm
  theta  = INTEG(omega, th0)
MACRO END
```

As in the case of the electrical circuits of Chapter 3, we see that the same physical device with emphasis on the same physical phenomena can lead to quite different model equations depending on the environment in which the model is being used. We have learned before that macros aren't really modular with respect to the incorporated *data structures*. Now, we learn that macros aren't even modular with respect to the represented *program structures*. The same physical device calls for quite different macros depending on the environments in which it is supposed to operate. The simplest "macro" representing an electrical resistor, for instance, must be stored in the macro library in two different versions, one modeling the equation:

$$u_R = R \cdot i_R \tag{5.38a}$$

and the other modeling the equation:

$$i_R = \frac{u_R}{R} \tag{5.38b}$$

If the resistor is placed over a current source, the current i_R through the resistor is known, and we need to use the macro which reflects the model according to eq(5.38a), whereas if we place the resistor over a voltage source, the voltage u_R across the resistor is known, and we need to use the macro which reflects the model according to eq(5.38b). Obviously, an *equation sorter* is insufficient. We require an *equation solver* which accepts general equalities of the type:

$$< expression > = < expression > \tag{5.39a}$$

or:

$$< expression > = 0.0 \tag{5.39b}$$

and which can solve these equalities for arbitrary variables. In the sequel, we shall discuss a software which satisfies this requirement.

However, let us return once more to the Ward–Leonard example. We now have two separate "macros" to describe the two DC–motors and to describe the DC–generator. Let us try to call them from an ACSL program and see what happens. Well, it won't work. According to our "models", ω_{12} is supposed to be a state variable of the DC–motor M_1, but at the same time also a state variable of the DC–generator G_2. This obviously can't be true. We just detected a *degeneracy* (structural singularity) of our system. Similarly, the armature current i_{a23} is supposedly a state variable of the DC–generator G_2, but at the same time also a state variable of the DC–motor M_3. So, we just found a second degeneracy of our system.

What can we learn from this example? While we can propose the implementation of an *automated single equation solver* as an extension to the common *equation sorter* to take care of the problem illustrated in eq(5.38a) and eq(5.38b), even such a mechanism won't suffice.

Algebraic loops are quite common, and they cut right across the borders between individual macros, i.e., the *IMPL* block as presented in Chapter 2 is totally incompatible with the demand of modular modeling. For this purpose, we should request a *compiler option* which would instruct the CSSL preprocessor to automatically generate the necessary *IMPL* block structures for us *after* the macro replacement and a partial equation sorting have already taken place. Unfortunately, even this feature won't help us with *system degeneracies* that occur as a result of subsystem coupling.

Let me repeat my conclusion from Chapter 3. We have exactly two choices. Either we put *state equations* as a mechanism to describe simulation models to the sword once and for all and use the *topological system description* directly for the simulation, or we come up with a *much* more powerful mechanism to generate appropriate state equations out of the topological system description than our simple macro facility represents. Let me develop this second path a little further in the remainder of this chapter.

5.6 Modular State–Space Models

As we have seen, it is necessary to store macros at all times in source form, and it is essential that the macro handler be executed *before* anything else happens to the simulation program. Therefore, it is not really essential that a CSSL provides for a macro handler of its own. It would be equally acceptable to employ a totally *independent* general purpose macro handler as a separate program to be called *before* the simulation compiler is entered. In this section, a new tool, DYMOLA [5.5], will be presented which is a stand–alone program that can be used as a front end to several different simulation languages. Two different versions of DYMOLA have been written. One is coded in Pascal, the other is coded in Simula. The Simula version runs on UNIVAC computers, the Pascal version runs on VAX/VMS and also on PC compatibles using Turbo Pascal (Version 4.0 or higher). DYMOLA is a *program generator* since a compiler switch decides for what simulation language code is to be generated. DYMOLA currently supports the simulation languages DESIRE [5.13] and SIMNON [5.4] (another direct executing language) and also plain Fortran, and an interface to ACSL [5.16] is currently under development. DYMOLA is not a *simulation language* in its own right since it does not provide for a simulation engine of its own. Instead, DYMOLA is a *modeling language* since it supports the user in coding more readable and better modularized hierarchically structured model descriptions. We shall dwell more on this topic in Chapter 15 of this text. On a first glance, DYMOLA looks like a powerful *macro handler.*

Let us discuss how the above DC–motor example can be coded in DYMOLA:

```
model type DCMOT
   terminal theta, omega, ua, uf, tauL, JL
   local ia, if, ui, psi, taum, Twist
   parameter Ra, Rf, kmot, Jm
   parameter La = 0.0, Lf = 0.0, Bm = 0.0
   default ua = 25.0, uf = 25.0
      Lf*der(if) = uf - Rf * if
      La*der(ia) = ua - ui - Ra * ia
      psi        = kmot * if
      taum       = psi * ia
      ui         = psi * omega
      der(Twist) = taum - tauL - Bm * omega
      Twist      = (Jm + JL) * omega
      der(theta) = omega
end
```

This code is fairly self–explanatory. However, let us discuss some of the special properties of DYMOLA model descriptions.

(1) DYMOLA variables belong either to the type *terminal* or to the type *local.* They are of type terminal if they are supposed to be *connected* to something outside the model. They are local if they are totally connected inside the model.

(2) Terminals can be either *inputs* or *outputs.* What they are, often depends on the environment to which they are connected. However, the user can explicitly specify what s/he wants them to be by *declaring* them as **input** or **output** rather than simply as **terminal.**

(3) Terminals can have *default values.* In this case, they don't need to be externally connected.

(4) DYMOLA constants can be declared to be of type *parameter*. For parameters, values can be assigned from outside the model. Parameters can have default values in which case it is not necessary to assign a value to them from outside the model.

(5) Derivatives are either expressed using the **der(.)** operator or a prime ($'$). It is also allowed to use a **der2(.)** operator or a double prime ($''$) to denote a second derivative, and even higher derivatives are admissible. Contrary to most CSSL's, DYMOLA allows us to use these operators anywhere in the equation, both to the left and to the right of the equal sign.

(6) Consequently, it is not possible to set initial conditions for the integrators inside a model which is clearly a disadvantage of DYMOLA.

(7) DYMOLA equations use the syntax of eq(5.39a). During the process of *model expansion*, equations are *solved* for the appropriate variable. For this reason, the SAL rule no longer applies. It is perfectly acceptable to have **der***(Twist)* on the left hand side of one equation, and *Twist* on the left hand side of another.

(8) Terms which are multiplied by a zero parameter are automatically eliminated during the model expansion. Consequently, if $La = 0.0$, the model equation $La * \mathbf{der}(ia) = ua - ui - Ra * ia$ is first replaced by the modified model equation $0.0 = ua - ui - Ra * ia$ which then results in one of three simulation equations, namely (i) $ua = ui + Ra * ia$, (ii) $ui = ua - Ra * ia$, or (iii) $ia = (ua - ui)/Ra$ depending on the environment in which the model is used. However, if $La \neq 0.0$, the model equation is always transformed into the simulation equation $\mathbf{der}(ia) = (ua - ui - Ra * ia)/La$. This is a very elegant way to solve the "variant macro" problem of ACSL.

(9) The above rule indicates that parameters with value 0.0 are treated in a completely different manner from all other parameters. This decision has a side effect. Parameters that are not set equal to zero are preserved in the generated simulation code, and can be interactively altered through the simulation program directly without a need to return to DYMOLA. Parameters with value 0.0 are optimized away by the DYMOLA compiler, and are not represented in the simulation code. However, the advantages of this decision are overwhelming, since this does away with an entire class of ***structural singularities***.

The above model can then be called in the following way:

$$\textbf{submodel}\ (DCMOT)\ dcm1(Ra = 2.0,\ Rf = 5.5,\ kmot = 1.0,\ Jm = 15.0)$$

It can be *connected* to the outside world using a *dot–notation*:

$$\begin{aligned} dcm1.ua &= kalph * err \\ dcm1.uf &= 12.0 \\ dcm1.JL &= crl1.JL \\ dcm1.tauL &= crl1.tauL \\ crl1.omega &= dcm1.omega \end{aligned}$$

where *crl1* is the name of a model of the cable reel type.

DYMOLA models are much more modular than ACSL macros since equations are automatically solved during model expansion for the variable which is appropriate in the context of the model call environment. The utilization of *named parameters* instead of *positional parameters* upon invocation of a DYMOLA model helps with long parameter lists. Default values can and should be assigned to many parameters, and with the named parameter convention, the user can selectively specify values for those parameters only for which the default values are not appropriate. The connection mechanism as presented so far is very general, although a little clumsy. Each connection corresponds to connecting two points of a circuit with a wire.

It can be noticed that wires are frequently grouped into cables or buses. For example, consider an RS232 connector. The RS232 male connector has 25 pins, while the corresponding RS232 female connector has 25 holes. It seems natural that a modeling language should provide for an equivalent mechanism. DYMOLA does this by providing so–called *CUTs*.

Let us look at the cable reel example once more. It can be noticed that the cable reel and the DC–motor have three variables in common, namely *omega*, *tauL*, and *JL*. We can therefore go ahead and declare those three variables in a *cut* rather than as simple *terminals*. The modified model type *DCMOT* looks now as follows:

```
model type DCMOT
   terminal theta, ua, uf
   cut mech(omega, tauL, JL)
   local ia, if, ui, psi, taum, Twist
   parameter Ra, Rf, kmot, Jm
   parameter La = 0.0, Lf = 0.0, Bm = 0.0
   default ua = 25.0, uf = 25.0
      Lf*der(if) = uf - Rf * if
      La*der(ia) = ua - ui - Ra * ia
      psi        = kmot * if
      taum       = psi * ia
      ui         = psi * omega
      der(Twist) = taum - tauL - Bm * omega
      Twist      = (Jm + JL) * omega
      der(theta) = omega
end
```

If we declare a similar cut in the model type *CABREL*, we can invoke in the main program a DC-motor *dcm1* of type *DCMOT*, and a cable reel *crl1* of type *CABREL*, and connect the cut *mech* of *dcm1* at the cut *mech* of *crl1*. This is coded as follows:

```
submodel (DCMOT) dcm1(Ra = ...)
submodel (CABREL) crl1(Bl = ...)
connect dcm1:mech at crl1:mech
```

The *connect statement* automatically generates the three model equations:

$$dcm1.omega = crl1.omega$$
$$dcm1.tauL = crl1.tauL$$
$$dcm1.JL = crl1.JL$$

Cuts can be hierarchically structured. For example, we could modify the model type *DCMOT* once more:

```
model type DCMOT
   terminal theta
   cut mech(omega, tauL, JL)
   cut elect(ua, uf)
   cut both[mech, elect]
   local ia, if, ui, psi, taum, Twist
   parameter Ra, Rf, kmot, Jm
   parameter La = 0.0, Lf = 0.0, Bm = 0.0
   default ua = 25.0, uf = 25.0
      Lf*der(if) = uf - Rf * if
      La*der(ia) = ua - ui - Ra * ia
      psi        = kmot * if
      taum       = psi * ia
      ui         = psi * omega
      der(Twist) = taum - tauL - Bm * omega
      Twist      = (Jm + JL) * omega
      der(theta) = omega
end
```

in which case we can either connect the cut *mech* and the cut *elect* separately, or we can connect *both* together. During expansion of the *connect* statement, DYMOLA checks that the connected cuts are structurally compatible with each other.

However, in many cases, even this won't suffice. We may notice that, by connecting a wire between two points in an electrical circuit, we actually connect *two* variables, namely the *potential* at the two points, and also the *current* that flows through the new wire. However, the two connections work differently. This is illustrated in Fig.5.12.

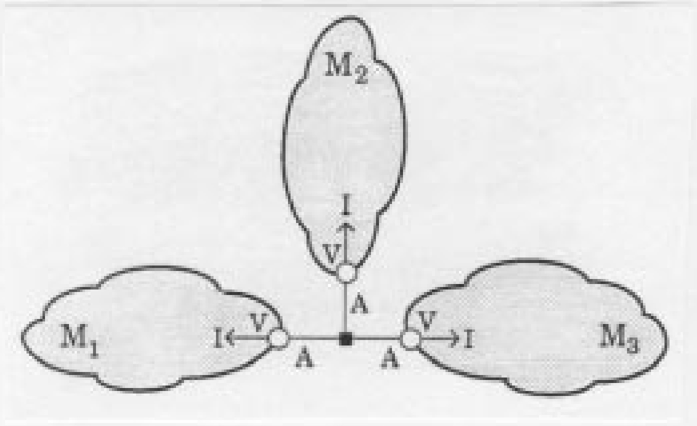

Figure 5.12. Connection conventions in an electrical circuit

While the potentials of all cuts that are connected at a point must be equal, the currents must add up to zero. Variables of type potential are called *across variables*, while variables of type current are called *through variables.* DYMOLA provides also for this second type of connection. The generalized form of a DYMOLA cut looks as follows:

```
cut <cut_name> (<through_variables> / <across_variables>)
```

If the three models $m1$, $m2$, and $m3$ have each a cut of type A declared as:

```
cut A(V/I)
```

we can use the connect statement:

```
connect m1:A at m2:A at m3:A
```

which will generate the following model equations:

$$\begin{aligned} m1.v &= m2.v \\ m2.v &= m3.v \\ m1.i + m2.i + m3.i &= 0.0 \end{aligned}$$

Notice that currents at cuts are normalized to point *into* the subsystem. If a current is directed the opposite way, it must take a minus sign on the cut definition.

This concept is more generally useful that just for electrical circuits. In a mechanical system, all positions, velocities, and accelerations are across variables, while all forces and torques are through variables. In a hydraulic system, water level and pressure are across variables, while water flow is a through variable. In a thermic system, temperature is an across variable, while heat flow is a through variable, etc. These similarities between different types of physical systems are particularly emphasized in the bond graph modeling methodology which will be discussed in Chapter 7 of this text.

One cut can be declared as the *main cut*. The main cut is the *default* cut in a connection, i.e. it suffices to specify the model name to connect the main cut of a submodel.

Sometimes it is useful to allow connections to take place *inside* a model instead of across model boundaries. For this purpose, DYMOLA provides a *node* declaration. Nodes are named, and cuts can be connected to nodes. Nodes are hierarchically structured the same way cuts are.

```
model M
   cut A(v1, v2), B(v3, v4), C(v5, v6)
   main cut D[A, B, C]
      ...
end
...
node N
connect M at (N, N, N)
```

The connect statement is equivalent to:

```
connect M:A at N, M:B at N, M:C at N
```

which is identical to saying:

```
connect M:A at M:B at M:C
```

which will result in the following set of equations:

$$\begin{aligned} M.v1 &= M.v3 \\ M.v3 &= M.v5 \\ M.v2 &= M.v4 \\ M.v4 &= M.v6 \end{aligned}$$

Sometimes, it is also useful to connect a type of variable through from a source to a destination. For this purpose, DYMOLA allows us to declare a directed *path* from an input cut to an output cut.

Let us assume we have a model describing a pump which is declared as follows:

```
model pump
   cut inwater(w1), outwater(w2)
   path water < inwater - outwater >
      ...
end
```

Let us assume we have two more models describing a pipe and a tank with compatibly declared cuts and paths, then we can connect the water flow from the pump through the pipe to the tank with the statement:

```
connect (water) pump to pipe to tank
```

One path can always be declared as the *main path.* If the main path is to be connected, the path name can be omitted in the connect statement.

Besides the *at* and *to* operators, DYMOLA provides some additional connection mechanisms which are sometimes useful. The *reversed* operator allows us to connect a path in the opposite direction. The *par* operator allows for a parallel connection of two paths, and the *loop* operator allows us to connect paths in a loop. Also, DYMOLA provides for abbreviations of some of these operators. The "=" symbol can be used as an alternative to *at*, the "−" operator can be used instead of *to*, the "//" operator can be used as an alternative to *par*, and the "\" operator denotes reversed. Examples of the use of these elements will be presented in Chapter 6.

5.7 The Equation Solver

DYMOLA can solve equations for any variable which appears *linearly* in the equation. This does not mean that the equation as a whole must be linear. For instance, DYMOLA is able to handle the following equation:

$$7 * x + y * y - 3 * x * y = 25 \tag{5.40}$$

if the variable it wants to solve this equation for is x. In this case, DYMOLA will transform the above equation into:

$$x = (25 - y * y)/(7 - 3 * y) \tag{5.41}$$

However, it cannot solve eq(5.40) for the variable y.

For some simple cases, it would be very easy to implement the appropriate transformation rules to handle even non–linear equations, but most non–linear equations don't provide for unique solutions. For example, the problem:

$$x^2 + y^2 = 1 \tag{5.42}$$

when solved for y has the two solutions:

$$y = +\sqrt{1 - x^2} \tag{5.43a}$$
$$y = -\sqrt{1 - x^2} \tag{5.43b}$$

DYMOLA would have no way of knowing which of the two solutions to use. The same is true when the non–linear equation is solved numerically by automatically generating an *IMPL* block around the equation. The numerical algorithm will simply approach one of the two solutions, often even depending on the chosen initial value, and that may be the wrong one. This will be illustrated by means of the following equation:

$$x^2 - 5x + 2 = 0 \tag{5.44}$$

We could reformulate this problem as follows:

$$x = \sqrt{5x - 2} \tag{5.45}$$

We now choose an initial value for x, and plug it iteratively into eq(5.45) until convergence. Fig.5.13 illustrates what happens in this case.

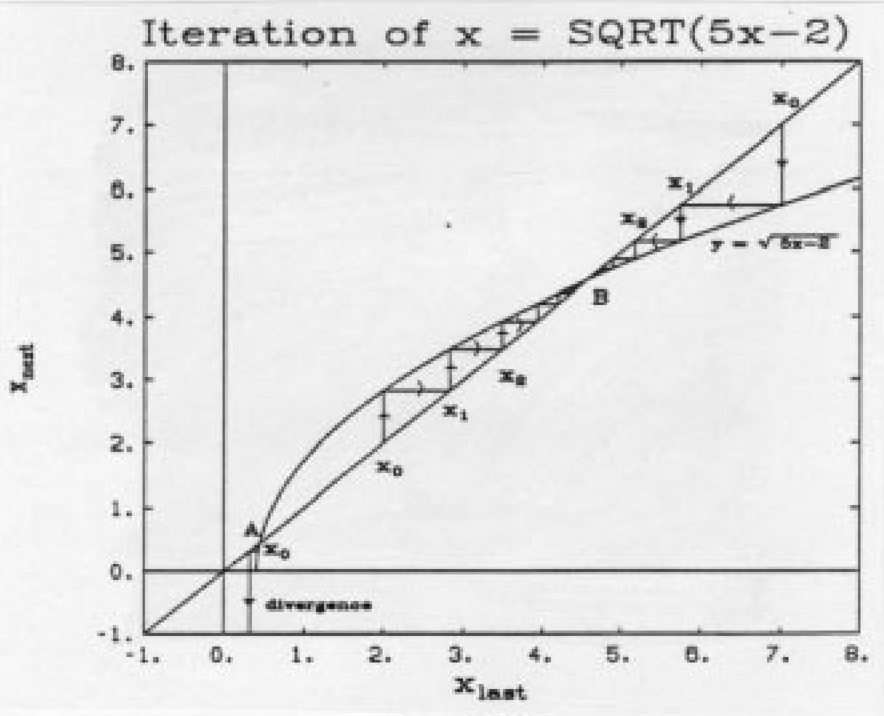

Figure 5.13. Iteration of non–linear equation

As can be seen, two solutions to this non–linear equation exist. Depending on the starting value, we either approach solution B, or the algorithm diverges. Solution A is an *unstable solution* of this iteration process.

However, we could have decided to formulate the problem in a different way:

$$x = 0.2x^2 + 0.4 \tag{5.46}$$

and iterate this equation instead. Fig.5.14 shows what happens in this case.

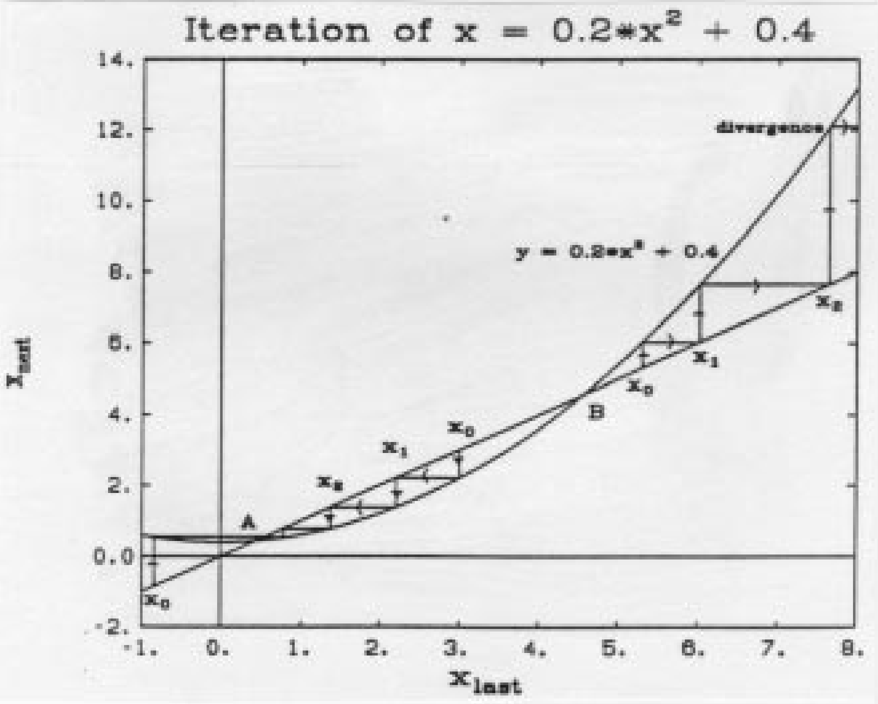

Figure 5.14. Iteration of non–linear equation

Obviously, this problem must have the same two solutions *A* and *B* as the previous one. However, as can be seen, this time, solution *A* is the stable solution, while solution *B* is unstable. Also, the range of attraction (i.e., the set of stable initial conditions) is different from the previous situation.

No generally applicable algorithm can be found for the automated solution of non–linear equations. The best that can probably be achieved is that the DYMOLA preprocessor stops when it comes across a non–linear equation, and requests help from the user. It may then store this information away for later reuse in another compilation of the same model. One possible answer that the user may provide is to request the system to build an *IMPL* block around the equation, and tell it which initial value to use for the iteration.

5.8 Code Optimization

DYMOLA already provides for a feature to eliminate trivial equations of the type $a = b$, by eliminating one of the two variables, and replace other occurrences of this variable in the program by the retained variable. This can significantly speed up the execution of the simulation program. However, much more could be done. It is foreseen to enhance the code optimizer by a fully automated algorithm to move all *constant computations* from the simulation language's "DYNAMIC" section into its program control section, and, in the case of DESIRE, all *output computations* from the "DYNAMIC" section into the "OUT" section. The code optimization is somewhat target language specific.

5.9 Linear Algebraic Loops

Let us revisit the electrical circuit as presented in Fig.3.12. The resulting set of equations was given in eq(3.34a-i). Let us discuss what DYMOLA would do with these equations.

Let us start from the initial set of equations:

$$u1 = R1 * i1 \tag{5.47a}$$
$$u2 = R2 * i2 \tag{5.47b}$$
$$u3 = R3 * i3 \tag{5.47c}$$
$$uL = L * \mathrm{der}(iL) \tag{5.47d}$$
$$U0 = u1 + u2 \tag{5.47e}$$
$$u3 = u2 \tag{5.47f}$$
$$uL = u1 + u2 \tag{5.47g}$$
$$i0 = i1 + iL \tag{5.47h}$$
$$i1 = i2 + i3 \tag{5.47i}$$

which are nine equations in the nine unknowns: $u1$, $u2$, $u3$, uL, $i0$, $i1$, $i2$, $i3$, and iL. $U0$ is not an unknown since this is the input to the system.

DYMOLA will start by recognizing that eq(5.47d) contains a **der(.)** operation, which determines that this equation must be solved for the derivative, and which moves iL from the list of unknowns to the list of knowns. It further recognizes that uL and $i0$ appear only in one of the remaining equations each, namely eq(5.47g) and eq(5.47h). This moves the variables uL and $i0$ from the list of unknowns to the list of knowns. Finally, it is recognized that eq(5.47f) is a trivial equation. We solve it for $u3$, and simultaneously replace all other occurrences of $u3$ by $u2$. Therefore, we have meanwhile the following set of *solved* equations:

$$\mathbf{der}(iL) = uL/L \tag{5.48a}$$
$$u3 = u2 \tag{5.48b}$$
$$uL = u1 + u2 \tag{5.48c}$$
$$i0 = i1 + iL \tag{5.48d}$$

Five equations remain:

$$u1 = R1 * i1 \tag{5.49a}$$
$$u2 = R2 * i2 \tag{5.49b}$$
$$u2 = R3 * i3 \tag{5.49c}$$
$$i1 = i2 + i3 \tag{5.49d}$$
$$U0 = u1 + u2 \tag{5.49e}$$

which depend on the five unknowns $u1$, $u2$, $i1$, $i2$, and $i3$. Each remaining equation contains at least two unknowns, and each of the unknowns appears in at least two equations. Thus, we have an algebraic loop.

At this moment, DYMOLA is stuck. However, this will change soon. DYMOLA could easily recognize that all unknowns appear linearly in all the remaining equations, and thus rewrite the above system of equations in a matrix form as:

$$\begin{pmatrix} 1 & 0 & -R1 & 0 & 0 \\ 0 & 1 & 0 & -R2 & 0 \\ 0 & 1 & 0 & 0 & -R3 \\ 0 & 0 & 1 & -1 & -1 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} u1 \\ u2 \\ i1 \\ i2 \\ i3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ U0 \end{pmatrix} \tag{5.50}$$

which could be coded in the following way:

```
Z09999 = [ 1, 0, -R1,   0,   0;  ...
           0, 1,  0,  -R2,   0;  ...
           0, 1,  0,    0, -R3;  ...
           0, 0,  1,   -1,  -1;  ...
           1, 1,  0,    0,   0]
[u1; u2; i1; i2; i3] =inv(Z09999)*[0; 0; 0; 0; U0]
```

Notice the notation. The square bracket denotes the beginning of a matrix definition. The "," operator separates elements in neighboring columns, while the ";" operator separates elements in neighboring rows.

If all we have here is a *linear algebraic loop*, the matrix $Z09999$ will be non–singular, and the above set of matrix equations has a unique solution.

At this point, the *code optimizer* can become active and recognize that all elements within $Z09999$ are constants, i.e., that the evaluation of the matrix can be moved out of the *DYNAMIC* section into the control block, and that even **inv**($Z09999$) is a constant expression which can be moved out into the control section where the matrix inversion will be performed exactly once prior to the execution of the simulation run.

Since DESIRE [5.13] is able to handle matrix expressions elegantly and very efficiently, this will be an easy task to implement. Currently, the regular version of DESIRE handles matrices only within the *interpreted* control section, and not within the *compiled DYNAMIC* block. However, a modified version DESIRE/NEUNET [5.14] exists already which was designed particularly for the simulation of *neural networks*. This version handles matrix expressions within the *DYNAMIC* block in a very efficient manner.

5.10 Non–linear Algebraic Loops

The problem with non–linear algebraic loops is exactly the same as with the solutions of single non–linear equations. Depending on how we iterate the set of equations, we may end up with one solution, or another, or none at all. Unfortunately, no generally applicable method can be found that would deal with this problem once and for all. As before, the best that DYMOLA may be able to do is interrupt the compilation, display the set of coupled algebraic equations on the screen together with the set of unknowns contained in these equations, and ask for help. Proper help may not always be easy to provide.

One way to tackle this problem is to get away from state–space models altogether. Instead of solving the set of first order differential equations:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{5.51}$$

we could try to find integration algorithms which can solve the more general set of equations:

$$\mathbf{A} \cdot \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{5.52}$$

directly, where **A** is allowed to be a singular matrix. This formulation takes care of all linear algebraic loops. Integration algorithms for this type of problems have been known for quite a while. Non–linear algebraic loops can be handled by integration algorithms that are able to solve the following set of implicit differential equations directly:

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u}, t) = 0.0 \tag{5.53}$$

The price to be paid for this generality is a reduction in execution speed and in solution robustness. It can no longer be guaranteed that these equations have exactly one correct solution for each set of initial conditions. MODEL [5.17] is a language in which this approach was implemented. MODEL is another experimental language with a user surface that is quite similar to the one of DYMOLA.

5.11 Structural Singularities

Usually, each component of a system that can store energy is represented by one or more differential equations. Capacitors and inductors of electrical circuits can store energy. Each capacitor and each inductor normally gives cause to one first order differential equation. Mechanical masses can store two forms of energy, potential energy and kinetic energy. Each separately movable mass in a mechanical system usually gives rise to a second order differential equation which is equivalent to two first order differential equations. However sometimes, this is not so. If we take two capacitors, and connect them in parallel, the resulting system order is still one. This is due to the fact that a linear dependence exists between the two voltages over the two capacitors (they are the same), and thus, they do not both qualify for state variables.

Such situations are called *system degeneracies* or *structural singularities.* Usually, subsystems will be designed such that no such singularities occur. The two parallel capacitors are simply represented in the model by one equivalent capacitor with the value:

$$C_{eq} = C_1 + C_2 \tag{5.54}$$

However, difficulties occur when subsystems are *connected* together, and when the structural singularity is a direct result of the coupling of the two subsystems. Let us assume the two subsystem orders of subsystems S_1 and S_2 are n_1 and n_2. If the coupled system S_c has a system order n_c which is smaller than the sum of n_1 and n_2, a structural singularity exists which is a result of the subsystem coupling.

These problems must be carefully analyzed, and they are often quite difficult to circumvent. For this purpose, let us study once more our Ward–Leonard group from before. The goal is to come up with a model of a DC–motor/generator which is powerful enough to be used under all circumstances, i.e., it must be possible to model the Ward–Leonard group by coupling together three submodels of the same type *DCMOT.*

Let us start by looking at the coupling between the motor M_1 and the generator G_2. Under this coupling, the two angular velocities are forced to be the same except for their signs which are opposite since the two machines are coupled back to back:

$$\omega_2 = -\omega_1 \tag{5.55}$$

Since the angular velocities of both submodels are essentially outputs of integrators, a structural singularity has occurred. We can now realize that the torque produced by each of the two rotating machines represents the torque load of the other machine, and that the inertia of each of the two machines is seen as an inertial load by the other machine. Therefore, each machine can contain the equations:

$$\frac{dT}{dt} = \tau_m - \tau_{load} - B_m\,\omega \tag{5.56a}$$

$$\omega = T/(J_m + J_{load}) \tag{5.56b}$$

and the couplings can be expressed as follows:

$$\tau_{load\_1} = \tau_{m\_2} - B_{m\_2}\,\omega_2 \tag{5.57a}$$
$$\tau_{load\_2} = \tau_{m\_1} - B_{m\_1}\,\omega_1 \tag{5.57b}$$
$$J_{load\_1} = J_{m\_2} \tag{5.57c}$$
$$J_{load\_2} = J_{m\_1} \tag{5.57d}$$

Now, we realize that we don't even need to specify the equality of the angular velocities any longer since they will be guaranteed automatically. In the coupled model, we waste computing time since we integrate the same angular velocity twice, but the structural singularity has been avoided.

Let us now look at the coupling between the generator G_2 and the motor M_3. We could express the coupling through a connection of the two armature cuts (u_a/i_a), but we immediately realize that the two currents, which are now forced to be equal except for their signs, are outputs of two integrators.

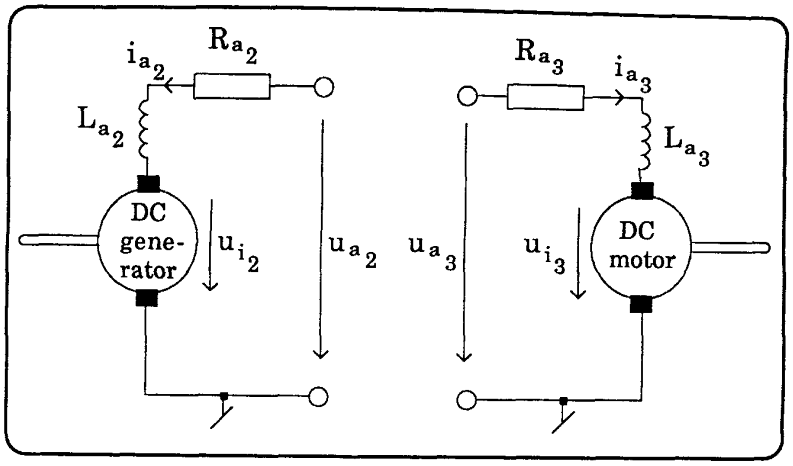

Figure 5.15. Electrical coupling in Ward–Leonard group

Thus, also this coupling represents a structural singularity. Fig.5.15 depicts the conventions for voltage and current directions. Therefore, we can write down the following equation:

$$ui_2 + Ra_2\ ia_2 + La_2 \frac{dia_2}{dt} = ui_3 + Ra_3\ ia_3 + La_3 \frac{dia_3}{dt} \tag{5.58}$$

which can be written in terms of ia_2 as:

$$\frac{dia_2}{dt} = (u_{load\_2} - ui_2 - Ra_2\ ia_2)/(La_2 + L_{load\_2}) \tag{5.59a}$$
$$u_{load\_2} = ui_3 + Ra_3\ ia_3 \tag{5.59b}$$
$$L_{load\_2} = La_3 \tag{5.59c}$$

or in terms of ia_3:

$$\frac{dia_3}{dt} = (u_{load\_3} - ui_3 - Ra_3\ ia_3)/(La_3 + L_{load\_3}) \tag{5.60a}$$
$$u_{load\_3} = ui_2 + Ra_2\ ia_2 \tag{5.60b}$$
$$L_{load\_3} = La_2 \tag{5.60c}$$

In other words, we can solve the structural singularity in exactly the same way as in the case of the mechanical coupling, again at the expense of some extra computation since the above equations eq(5.59a-c) and eq(5.60a-c) represent the same physical variables. Therefore, we can write the following equations into our generic *DC-MOT* model:

$$\frac{di_a}{dt} = (u_{load} - u_i - R_a\ i_a)/(L_a + L_{load}) \tag{5.61a}$$
$$u_a = u_i + R_a\ i_a + L_a \frac{di_a}{dt} \tag{5.61b}$$

The coupling equations are written as follows:

$$u_{load\_2} = ui_3 + Ra_3\ ia_3 \tag{5.62a}$$
$$L_{load\_2} = La_3 \tag{5.62b}$$
$$u_{load\_3} = ui_2 + Ra_2\ ia_2 \tag{5.62c}$$
$$L_{load\_3} = La_2 \tag{5.62d}$$

Let us now look how these equations must be modified if it has been decided to ignore the armature inductances. In that case, in order to avoid an algebraic loop, the resistance of the other machine cannot be included in the load voltage u_{load}, but must instead be treated as a resistive load. The generic model equations now look as follows:

$$i_a = (u_{load} - u_i)/(R_a + R_{load}) \tag{5.63a}$$
$$u_a = u_i + R_a\, i_a \tag{5.63b}$$

and the coupling equations can be written as follows:

$$u_{load\_2} = ui_3 \tag{5.64a}$$
$$R_{load\_2} = Ra_3 \tag{5.64b}$$
$$u_{load\_3} = ui_2 \tag{5.64c}$$
$$R_{load\_3} = Ra_2 \tag{5.64d}$$

The following ACSL [5.16] macro is a general macro that can be used both as a DC–motor and as a DC–generator.

```
MACRO DCMOT(theta, omega, taum, Jm, ua, RLa, ia, ui, uf, uld, RLld, ...
  tauld, Jld, Ra, La, Rf, Lf, kmot, Jm0, Bm, flag, if0, ia0, T0, th0)
  MACRO redefine iadot, if, ifdot, psi
  MACRO redefine Twist, Tdot
  MACRO standval if0 = 0.0, ia0 = 0.0, T0 = 0.0
  MACRO standval th0 = 0.0
  MACRO if (flag = IND) labind
    if     = uf/Rf
    ia     = (uld - ui)/(Ra + RLld)
    ua     = ui + Ra * ia
    RLa    = Ra
  MACRO goto goon
  MACRO labind..continue
    ifdot  = (uf - Rf * if)/Lf
    if     = INTEG(ifdot, if0)
    iadot  = (uld - ui - Ra * ia)/(La + RLld)
    ia     = INTEG(iadot, ia0)
    ua     = ui + Ra * ia + La * iadot
    RLa    = La
  MACRO goon..continue
    psi    = kmot * if
    taum   = psi * ia
    ui     = psi * omega
    Tdot   = taum - tauld - Bm * omega
    Twist  = INTEG(Tdot, T0)
    omega  = Twist/(Jm + Jld)
    theta  = INTEG(omega, th0)
    Jm     = Jm0
 MACRO END
```

With this macro, we can simulate the Ward–Leonard group in the control configuration depicted in Fig.5.16.

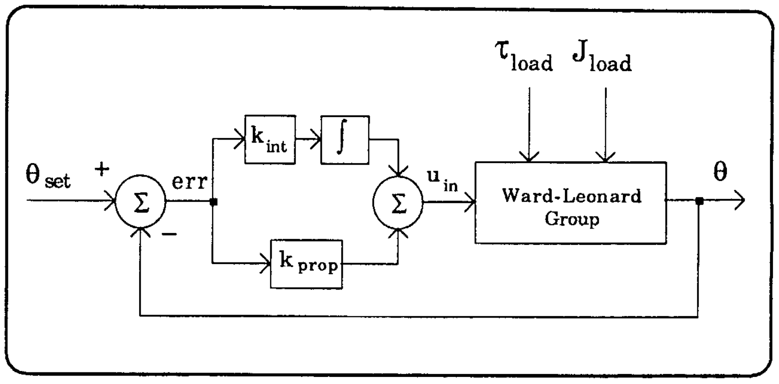

Figure 5.16. Block diagram of controlled Ward–Leonard group

The Ward–Leonard group is embedded in a position control circuit with a PI–controller. The angular position θ_{set} is set to 10.0 initially, and is reduced to 5.0 at time 50.0. The Ward–Leonard group is originally idle, but it is loaded both with a torque τ_{load} and an inertia J_{load} at time 100.0. The simulation extends over 200.0 time units. The following program implements this control problem.

```
PROGRAM Ward − Leonard group
  INITIAL
    MACRO DCMOT(...)
      ...
    MACRO END
    constant ...
      Jm0 = 0.05, Bm = 2.0E − 4, Ra = 10.0, La = 0.5E − 3, ...
      Rf = 25.0, Lf = 2.2E − 3, kmot = 0.5, kampl = 0.06, ...
      kint = 0.002, uf1 = 25.0, uf3 = 25.0, uld1 = 25.0, ...
      RLld1 = 0.0, tmx = 200.0
    cinterval cint = 0.5
    thset = 10.0
    Jld   = 0.0
    tauld = 0.0
    schedule angle .at. 50.0
    schedule load .at. 100.0
  END $ "of INITIAL"
```

```
        DYNAMIC
          DERIVATIVE
            err    = thset - theta
            uin    = kampl * err + kint * INTEG(err, 0.0)
            DCMOT(th1, om1, tm1, Jm1, ua1, RLa1, ia1, ui1 = ...
                   uf1, uld1, RLld1, tld1, Jld1, ...
                   Ra, La, Rf, Lf, kmot, Jm0, Bm, "NOIND")
            DCMOT(th2, om2, tm2, Jm2, ua2, RLa2, ia2, ui2 = ...
                   uf2, uld2, RLld2, tld2, Jld2, ...
                   Ra, La, Rf, Lf, kmot, Jm0, Bm, "NOIND")
            DCMOT(th3, om3, tm3, Jm3, ua3, RLa3, ia3, ui3 = ...
                   uf3, uld3, RLld3, tld3, Jld3, ...
                   Ra, La, Rf, Lf, kmot, Jm0, Bm, "NOIND")
            tld1   = tm2 - Bm * om2
            Jld1   = Jm2
            uf2    = -uin
            uld2   = ui3
            RLld2  = RLa3
            tld2   = tm1 - Bm * om1
            Jld2   = Jm1
            uld3   = ui2
            RLld3  = RLa2
            tld3   = tauld
            Jld3   = Jld
            theta  = th3
          END $ "of DERIVATIVE"
          DISCRETE angle
            thset  = 5.0
          END $ "of DISCRETE angle"
          DISCRETE load
            Jld    = 0.05
            tauld  = 0.01
          END $ "of DISCRETE load"
          termt (t.gt.tmx)
        END $ "of DYNAMIC"
      END $ "of PROGRAM"
```

The discontinuous driving functions were implemented using time–events.

The results of this simulation are shown in Fig.5.17.

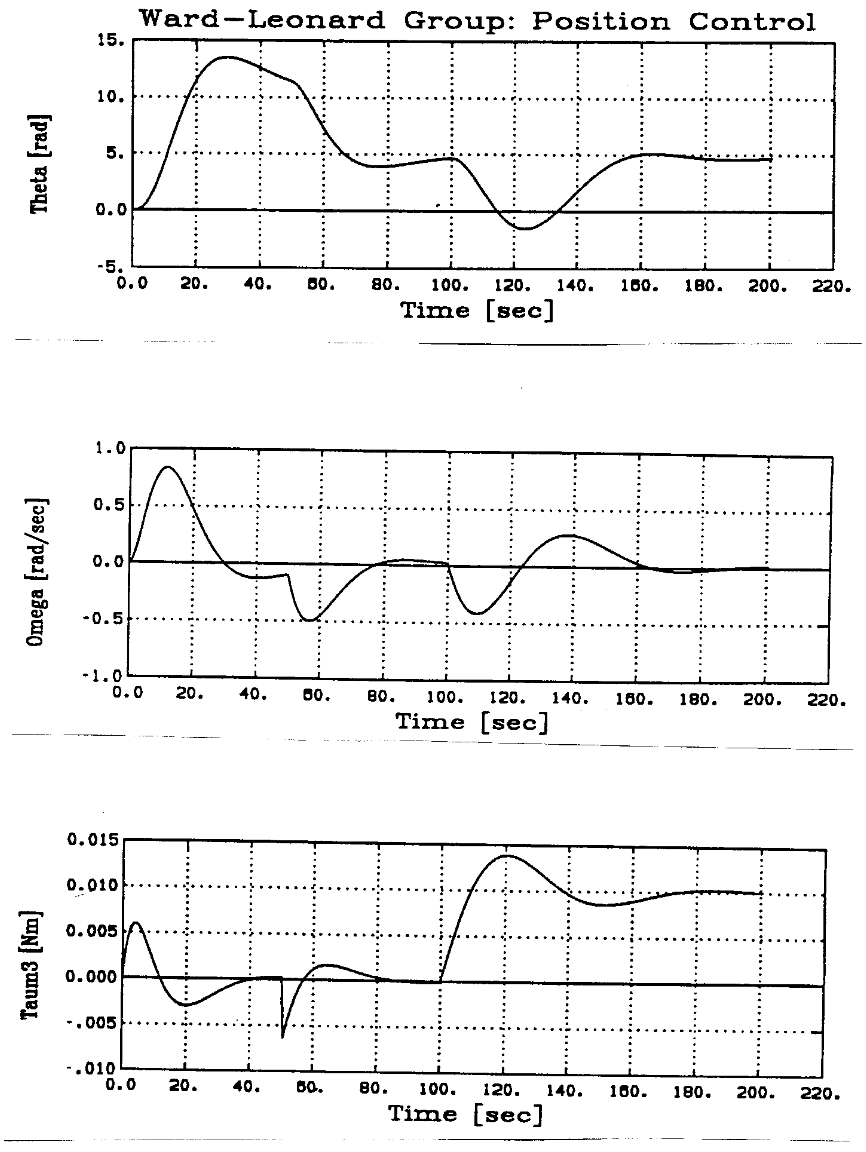

Figure 5.17. Simulation results of Ward–Leonard group

The output position θ_3 follows the input rather well. The overshoot behavior is due to the integral portion of the control circuit. The torque produced in motor M_3 is responsible for the rotation of the motor. Since the motor is idle, the torque returns to zero as soon as the desired position has been reached. However, after time $t = 100.0$, the motor is loaded with a torque load. Now, in order to keep the position stationary, the motor needs to produce a constant motor torque that compensates for the torque load. The integral portion of the controller is responsible for returning the angular position of the motor bias–free back to its desired value after each disturbance. Nevertheless, the group is not able to keep the position constant under the varying load. In order to improve the performance of the system, we must guarantee a constant rotation of the driving shaft ω_1. This can be achieved by replacing the driving motor M_1 with a stronger machine, or eventually, by replacing it with a synchronous AC–motor.

The modeling approach did work since the *DCMOT* macro does not call for many equations to be rearranged with respect to their outputs. However, quite a bit of insight was needed in order to get all equations into an adequate form, and to solve all problems related to system degeneracies and algebraic loops. The approach is therefore not very convenient and user–friendly, and it may be quite difficult to apply this solution to a more intricate problem than the one presented.

We are convinced that, in principle, the DYMOLA approach is the better answer to the problem. However, at this moment, DYMOLA won't do the job yet. A number of extensions will be needed before DYMOLA will be able to take care of such problems in a completely automated manner:

(1) DYMOLA should be able to eliminate variables not only from equations of the type $a = b$, but also from equations of the type $a + b = 0$.

(2) DYMOLA should be able to recognize equations that have been specified twice, and eliminate the duplicate automatically.

(3) DYMOLA should be able to handle superfluous connections, i.e., if we specify that $\theta_2 = -\theta_1$, it is obviously true that also $\omega_2 = -\omega_1$. However, DYMOLA won't let us specify this additional connection at the current time. Superfluous connections should simply be eliminated during the model expansion.

(4) DYMOLA should recognize that connections of outputs of integrators can always be converted into connections of inputs of

these integrators, i.e., if we have specified that $ia_3 = -ia_2$, it is obviously true that $iadot_3 = -iadot_2$. This reformulation can help to eliminate structural singularities, usually at the cost of generating additional algebraic loops.

(5) DYMOLA should be enabled to handle linear algebraic loops in the manner previously suggested.

A fair amount of program development and even research is still needed before DYMOLA can be turned into a production code, but we are convinced that this is a good way to go. We shall return to this problem once more in Chapter 15 of this text.

5.12 Large Scale System Modeling

A true disadvantage of the above proposed methodology is the fact that, as with macros, DYMOLA models need to be stored as source code. This may be quite impractical if models of maybe 20,000 lines of code are to be simulated. A small structural modification of one single equation within one single submodel will force us to recompile the entire code which may take quite a long time, and consume an undue amount of computing resources.

Separate compilation of submodels is difficult to achieve. One way to solve this problem is to generate the target code such that each single equation is preceded by a label and followed by a "goto" statement with an address which is not static, but which is stored in a large connection table. If a submodel is to be modified, it suffices to recompile that submodel, and to correct the connection table accordingly. None of the systems presented so far offers this capability. One system that does offer this facility is SYSMOD [5.19]. The SYSMOD language is a superset of Pascal. The SYSMOD system consists of a Pascal–coded preprocessor that compiles (sub–)models into Fortran subroutines, a Fortran–coded simulation run–time system containing the integration routines and output routines, and a special purpose "linker" which is responsible for updating the connection table after a recompilation of a submodel has occurred.

The price for this separate submodel compilation capability is a reduction in run–time efficiency. According to information obtained from the producers of SYSMOD, the run–time overhead is about 20% which seems quite acceptable. We have not yet had a chance to verify or reject this information on the basis of significantly large sample programs.

5.13 Graphical Modeling

With the advent of increasingly powerful engineering workstations, the demand has risen to model systems graphically on the screen. Submodels are maintained in a *model library*. Such submodels could be either DYMOLA models, regular CSSL macros, transfer functions, linear state–space descriptions written in matrix form, or static characteristics. Each of these models is associated with an *icon* which is stored in an *icon library*. Invoking a submodel simply means to place the corresponding item on the screen. Connections between submodels are done by drawing a line between two terminals (cuts) of two icons.

Several such systems exist already on the software market. Some of these systems are *generic program generators* in that they allow the user to specify what code s/he wants to generate as a result of the *graphical compilation* (i.e., the evaluation of the graph). One such system, EASE+ [5.8], has been successfully employed as a graphical preprocessor to ACSL. Others are either stand–alone systems, such as EASY5 [5.1], or they are integral parts of particular software systems, such as SYSTEM–BUILD [5.11] which has been designed as a modeling tool for MATRIX$_X$ [5.10], or MODEL–C which is a modeling tool for CTRL–C [5.18].

Most of these systems are based on the concept of *block diagram modeling*. The basic building blocks are those used in block diagrams, i.e., single–input/single–output (SISO) system descriptions, summers, and branching points. The disadvantage of these systems is obvious. They do not provide for a hierarchical decomposition of data structures (cuts), and they do not provide for the representation of through variables.

Some systems are specialized tools for particular types of models. For example, Workview [5.20] is a graphical modeling system for electronic circuits. As a result of the graphical compilation, Workview generates a PSPICE [5.15] program. Workview will be presented in Chapter 6.

Another system, HIBLIZ [5.6], has been designed specifically as a graphical preprocessor to DYMOLA. HIBLIZ supports all concepts that DYMOLA does, and can therefore be used for generic modeling of arbitrarily coupled systems with hierarchical cuts and across as well as through variables. The result of the graphical compilation is a DYMOLA program. Unfortunately, while most of the available graphical systems have been developed for PC compatibles, HIBLIZ runs currently on Silicon Graphics (IRIS) machines only which makes the software much less accessible (though faster executing).

One problem with the graphical approach is the fact that the screen is not large enough to depict reasonably complex systems. One typical solution to this problem is to resort to a *virtual screen.* The virtual screen can be made arbitrarily large, and the actual screen covers a *window* out of the virtual screen, i.e., the physical screen can be moved over the virtual screen much like a short–sighted person may use a magnifying glass to move over a page of a book. Another approach is the *zoom in* facility which allows us to modify the size of the icons on the screen. Virtual screens and zooming are often combined. When zooming in on a portion of the virtual screen, the window of the physical screen is made smaller, i.e., it covers a smaller portion of the virtual screen.

Also in this respect, HIBLIZ offers a rather unique feature which is called a *breakpoint.* Breakpoints allow the programmer to alter the drawing as a function of the magnification (zooming). As an example, it is possible to start out with a box as large as the virtual screen which is empty except for the name of the problem which appears in large letters. The physical screen at this moment coincides with the virtual screen. As soon as we start zooming in on the graph, a breakpoint is passed, and suddenly, the title disappears and is replaced by some text which described the purpose of the model. When zooming in further, another breakpoint is passed, and the text is replaced by a set of smaller boxes with interconnections. Each box contains the name of the submodel represented by this box. Now, we can zoom in on any of these boxes. Meanwhile, the physical screen has become considerably smaller than the virtual screen, and we no longer see the entire picture at once. Again, a new breakpoint is passed, and now, we see a text that describes the purpose of the submodel on which we are currently focusing. When zooming in further, the text is replaced by the internal structure of the submodel which may consist of some more submodels with interconnections. Ultimately, we come to the layer of *atomic models* in which case the description may be replaced by a graph denoting a static characteristic, by a state–space model, by a transfer function, or by a set of DYMOLA statements. If we zoom in on the point where an interconnection meets a box, we can notice that each such point in reality is represented by a little box itself. Zooming in further on that little box, we can determine the nature of the cut that is represented by the interconnection, i.e., we can learn about the variables represented in the cut.

The graphical representation can also be used later, i.e., during or after the simulation phase. For example, HIBLIZ allows to point to a particular connection after the simulation has been executed. As a result of this action, a new window is opened in which the trajectories of all variables contained in the cut are displayed as functions of time.

On the long run, this is clearly the right approach. However, in order to enhance the accessibility of the HIBLIZ code, it is hoped that the developers of the software (the Technical University of Lund, Sweden) will port the code to *X Windows.* We shall talk more about HIBLIZ in Chapter 15 of this text.

It was demonstrated how hierarchical modeling has become the key issue to coping with the increasing demands of modern large scale continuous system simulation. I would like to acknowledge in particular the important research results obtained by Hilding Elmqvist of the Technical University at Lund. Hilding Elmqvist produced in 1975 the first *direct executing CSSL language*, SIMNON, which paved the way for the work that resulted much later in the DESIRE software which has been advocated in this text. SIMNON was developed by Hilding Elmqvist as his Master Thesis. Later in 1978, Hilding developed DYMOLA for his Ph.D. Dissertation, a software system which is still, twelve years later, very much state–of–the–art. In 1982, he developed the first prototype of HIBLIZ, at least three years before any competitor products came on the market. HIBLIZ is even today considerably more powerful than all of its competitors since it implements hierarchical cuts and through as well as across variables.

5.14 Summary

In this chapter, we have discussed the problems associated with large scale system modeling: modularity and hierarchical decomposition of submodels. We have introduced a new language, DYMOLA, which is particularly well suited to support the process of modeling large scale systems. In subsequent chapters, we shall present many examples of the concepts that were introduced here.

References

[5.1] Boeing Computer Services (1988), *Easy5/W — User's Manual*, Engineering Technology Applications (ETA) Division, P.O.Box 24346, MS 7L–23, Seattle, WA, 98124.

[5.2] François E. Cellier (1979), *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, Ph.D. Dissertation, ETH Zürich, Diss ETH No 6483, Dept. of Automatic Control, Swiss Federal Institute of Technology, CH–8092 Zürich, Switzerland.

[5.3] Olle I. Elgerd (1971), *Electric Energy Systems Theory: An Introduction*, McGraw–Hill, New York.

[5.4] Hilding Elmqvist (1975), *SIMNON - An Interactive Simulation Program for Non–linear Systems — User's Manual*, Report CODEN: LUTFD2/(TFRT–7502), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[5.5] Hilding Elmqvist (1978), *A Structured Model Language for Large Continuous Systems*, Ph.D. Thesis, Report CODEN: LUTFD2/(TRFT–1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[5.6] Hilding Elmqvist (1982), "A Graphical Approach to Documentation and Implementation of Control Systems", *Proceedings Third IFAC/IFIP Symposium on Software for Computer Control (SOCOCO'82)*, Madrid, Spain, Pergamon Press, Oxford, U.K.

[5.7] EPRI (1983), *Modular Modeling System (MMS): A Code for the Dynamic Simulation of Fossil and Nuclear Power Plants*, Report: CS/NP–3016–CCM, Electric Power Research Institute, 3412 Hillview Ave., Palo Alto, CA 94304.

[5.8] Expert-EASE Systems, Inc. (1988), *EASE+ — User's Manual*, 1301 Shoreway Rd., Belmont, CA 94002.

[5.9] IBM Canada Ltd. (1972), *Continuous System Modeling Program III (CSMP-III) — Program Reference Manual*, Program Number: 5734-XS9, Form: SH19-7001-2, IBM Canada Ltd., Program Produce Centre, 1150 Eglington Ave. East, Don Mills 402, Ontario, Canada.

[5.10] Integrated Systems, Inc. (1984), *Matrix$_x$ User's Guide, Matrix$_x$ Reference Guide, Matrix$_x$ Training Guide, Command Summary and On-Line Help*, 2500 Mission College Blvd., Santa Clara, CA 95054.

[5.11] Integrated Systems, Inc. (1985), *System Build User's Guide*, 2500 Mission College Blvd., Santa Clara, CA 95054.

[5.12] Dirk L. Kettenis (1988), *COSMOS - Reference Manual*, Dept. of Computer Science, Agricultural University Wageningen, Hollandseweg 1, 6706 KN Wageningen, The Netherlands.

[5.13] Granino A. Korn (1989), *Interactive Dynamic-System Simulation*, McGraw-Hill, New York.

[5.14] Granino A. Korn (1991), *Neural-Network Experiments on Personal Computers*, M.I.T. Press, Cambridge, MA.

[5.15] MicroSim Corp. (1987), *PSPICE User's Manual*, 20 Fairbanks Rd., Irvine, CA 92718.

[5.16] Edward E. L. Mitchell, and Joseph S. Gauthier (1986), *ACSL: Advanced Continuous Simulation Language — User Guide / Reference Manual*, Mitchell & Gauthier Assoc., 73 Junction Square, Concord, MA 01742.

[5.17] Thomas F. Runge (1977), *A Universal Language for Continuous Network Simulation*, Technical Report UIUCDCS-R-77-866, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL, 61801.

[5.18] Systems Control Technology, Inc. (1985), *CTRL-C, A Language for the Computer-Aided Design of Multivariable Control Systems, User's Guide*, 2300 Geng Rd., P.O.Box 10180, Palo Alto, CA 94303.

[5.19] Systems Designers plc (1986) *Sysmod User Manual*, Release 1.0, D05448/14/UM, Ferneberga House, Alexandra Rd., Farnborough Hampshire GU14 6DQ, U.K.

[5.20] Viewlogic Systems, Inc. (1988) *Workview Reference Guide*, Release 3.0, and: *Viewdraw Reference Guide*, Version 3.0, 313 Boston Post Rd. West, Marlboro, MA 01752.

[5.21] John V. Wait, and DeFrance Clarke, III (1976), *DARE-P User's Manual*, Version 4.1, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, AZ, 85721.

Homework Problems

[H5.1]* Control System

Fig.H5.1 shows a typical single input single output (SISO) control system designed in the frequency domain. A second order plant is controlled by use of a *lead compensator*. The output variable x is measured using measurement equipment with dynamic behavior of its own.

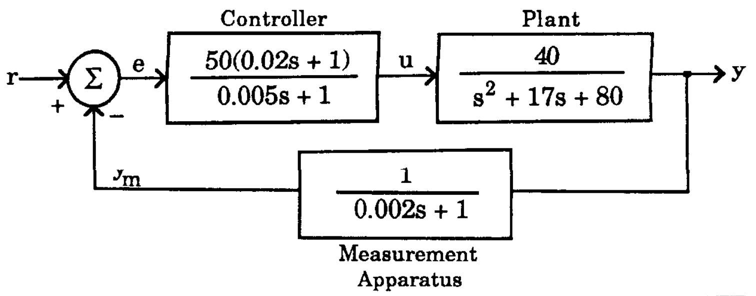

Figure H5.1. Block diagram of a SISO control system

Many CSSL's (such as ACSL) offer built in macros to model systems that are either totally or partially described in the frequency domain directly without need to transform the model back into the time domain. Model this system in ACSL using the *LEDLAG* macro to model the controller, the *TRAN* macro to model the plant, and the *REALPL* macro to model the measurement equipment. Simulate a step response of this system. Use a dynamic termination condition to bring the simulation to an end as soon as three values of y that are separated in time by $\Delta t = 0.03sec$ differ less than 0.001 from each other. Use ACSL's *DELAY* operator to construct the signals:

$$\Delta y_1 = y(t) - y(t - \Delta t); \quad \Delta y_2 = y(t - \Delta t) - y(t - 2\Delta t) \qquad (H5.1)$$

which can then be used in a *TERMT* condition.

Since this is a linear system, we could also simulate it in CTRL–C (or MATLAB) directly. Use CTRL–C's (MATLAB's) *TF2SS* function to transform each of the three transfer functions into state–space models. Thereafter use CTRL–C's *INTERC* function to come up with a state–space model for the total interconnected system. Thereafter, use CTRL–C's *SIMU* or *STEP* functions to simulate the system with step input. Remember that CTRL–C offers interactive HELP for all its functions. If you use MATLAB, consult the manual for the names of these functions. All CTRL–C capabilities are offered in MATLAB as well.

[H5.2]* Sampled Data Control System

Fig.H5.2 shows a sampled data control system [5.16]. The plant to be controlled is a second order "Type 1" system (indicating that the plant has a pole at the origin). This model represents a DC–motor in which the armature inductance has been neglected. The output variable x is the angular position of the motor. The system is to be controlled by a microprocessor. An optimal controller was designed in the s–domain, and it was found that a *lead compensator* of the form:

$$G_c = \frac{1 + 2.5s}{1 + 0.5s} \qquad (H5.2a)$$

would be optimal for this system using unity feedback (i.e., assuming an ideal measurement apparatus).

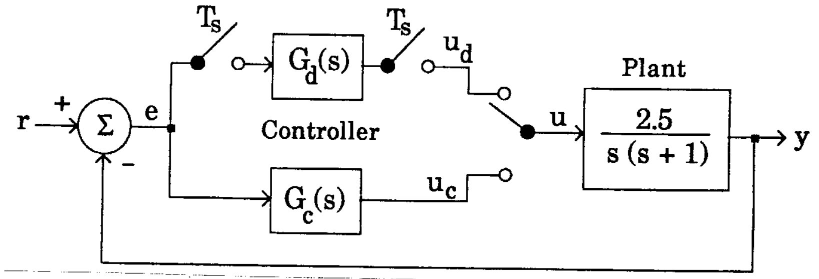

Figure H5.2. Block diagram of a sampled data control system

We want to simulate the continuous control system in ACSL using the *LEDLAG* macro to describe the controller, and the *TRAN* macro to describe the plant. Simulate the system over a duration of 5 *sec*.

Unfortunately, the above simulation is not very realistic since the microprocessor cannot properly represent the lead compensator dynamics. We can compute an equivalent discrete controller in CTRL–C (or MATLAB). Use the *TF2SS* function to get a state–space representation of your controller. Thereafter, use the *C2D* function to transform the continuous–time controller model to an equivalent discrete–time controller model. Use a sampling interval of T_s = 0.1 *sec*. It turns out that the transformation affects only the **A** matrix and the **b** vector (in our example, they are both scalars), while the output equation remains unchanged. Augment your previous ACSL program by including now the discrete controller using the design that was computed in CTRL–C (MATLAB). Use a *DISCRETE* block with an *INTERVAL* specification to model the discrete controller. This can be achieved using the following language construct:

```
DISCRETE controller
  INTERVAL Ts = 0.1
  PROCEDURAL
    xinew = a * xi + b * e
    udis  = c * xi + d * e
    xi    = xinew
  END
END
```

The *INTERVAL* specification will ensure that this block is being executed exactly once every T_s time units. The *PROCEDURAL* declaration is necessary to break the algebraic loop between ξ and ξ_{new}. Upon execution of the *DISCRETE* block, a new value of u_{dis} is computed which thereafter stays constant for the duration of one sampling period (sample and hold). Model the switch using ACSL's *FCNSW* function. Simulate the step response of the sampled data system, and compare the results to the idealized continuous–time simulation performed above.

Even the sampled data model is not truly realistic since it takes the computer some time to perform the computations expressed in the *DISCRETE* block. Assume that the time needed for the computation is $\Delta t = 0.003$ *sec*. This can be simulated by replacing the u_{dis} signal in the above described *DISCRETE* block by another signal, say u_{st}, and schedule from within the *DISCRETE* block another *DISCRETE* block called *dac* to be executed Δt time units in the future using ACSL's time–event scheduling facility. All that the *dac* block needs to do is to pass the current value of the u_{st} variable on to u_{dis}. Simulate the step response of the once more refined model, and compare the results with those of the previous simulations. If you run ACSL through CTRL–C or MATLAB, you can easily plot all simulation results on top of each other on one graph.

[H5.3] Water Flow Through a Reservoir

A community uses a small reservoir for irrigation purposes, but also to prevent damage otherwise produced by flooding during storms. When the reservoir contains a water volume of $V = V_D = 4000\ m^3$, water will begin to overflow from the reservoir into a series of drainage channels. Assume that, at $t = 0$, the reservoir contains $V_0 = 3900\ m^3$ of water. A storm that lasts for 25 *hours* adds water to the reservoir as specified in Table H5.3a.

Table H5.3a Water inflow $\dot{q}_{in}$ during a storm

time t $[h]$	inflow $\dot{q}_{in}$ $[m^3/h]$
0.0	10.0
1.0	35.0
2.0	58.0
3.0	70.0
4.0	75.0
5.0	68.0
6.0	55.0
7.0	38.0
8.0	28.0
9.0	21.0
10.0	18.0
11.0	16.0
12.0	14.0
13.0	13.0
14.0	12.0
15.0	11.0
20.0	10.5
25.0	10.0

The overflow is described by the function:

$$\dot{q}_{out} = 0.02 \cdot f(\dot{q}_{out}) \cdot [\max(V - V_D, 0.0)]^{1.5} \qquad (H5.3)$$

where the overflow characteristic $f(\dot{q}_{out})$ is specified in Table H5.3b.

Table H5.3b Water outflow characteristics $f(\dot{q}_{out})$

outflow $\dot{q}_{out}$ $[m^3/h]$	$f(\dot{q}_{out})$
0.0	0.8
25.0	0.85
50.0	0.95
75.0	1.0
100.0	1.0

Code this problem in ACSL using tabular functions and an *IMPL* block to describe the implicitly defined overflow characteristic.

Simulate this system over the duration of the storm, and plot on separate graphs the inflow rate, the outflow rate, and the volume content of the reservoir.

[H5.4] Surge Tank Simulation

A water turbine is fed from a reservoir through a pressurized pipe. The turbine in turn feeds an electrical generator (a synchronous AC–machine).

In case of a short–circuit on the electrical circuit, it may be necessary to shut down the generator very fast (within a few seconds) which may force us to close the valve in front of the turbine within a very short time period. When the valve is open, water flows from the reservoir into the turbine. This water flow contains a potentially destructive amount of *kinetic energy*. If we are forced to close the valve quickly, the kinetic energy could destroy the rear end of the pressure tunnel. In order to prevent such damage from happening, we need to add a surge tank to the system into which the water can escape. The surge tank will help to convert the kinetic energy into potential energy, thereby reducing the maximum pressure in the pressure tunnel. The system is shown in Fig.H5.4a.

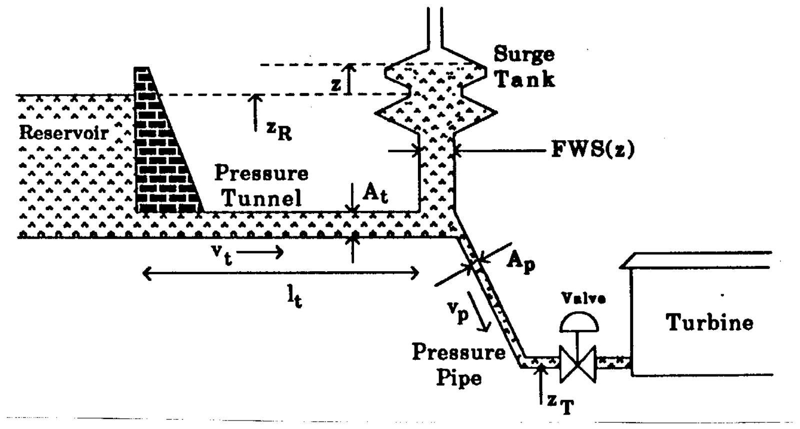

Figure H5.4a. Power generation using a water turbine

The reservoir surface is at an altitude of $z_R = 3168\ m$. It is assumed that the reservoir is sufficiently large so that the outflow of water from the reservoir does not noticeably change the water level in the reservoir.

We can compute the amount of water that flows from the reservoir into the pressure tunnel by formulating Newton's law for the pressure tunnel:

$$m\dot{v}_t = F_{in} - F_{out} - F_{Fr} \qquad (H5.4a)$$

F_{in} and F_{out} are the forces produced by the water pressure at the inflow and outflow of the pressure tunnel. These forces are equal to the water pressure multiplied by the active surface, i.e.

$$F_{in} = A_t \cdot p_{in} \qquad (H5.4b)$$
$$F_{out} = A_t \cdot p_{out} \qquad (H5.4c)$$

where the cross–section of the pressure tunnel A_t has a value of 12 m^2. The water pressures can be computed to be

$$p_{in} = \rho \cdot g \cdot h_{in} \tag{H5.4d}$$

$$p_{out} = \rho \cdot g \cdot h_{out} \tag{H5.4e}$$

where ρ is the water density ($\rho = 1000\ kg\ m^{-3}$), g is the gravity ($g = 9.81\ m\ sec^{-2}$), and h_{in} and h_{out} denote the momentary values of the water columns (i.e., h_{in} is the depth of the reservoir, and h_{out} is the current water level in the surge tank minus the altitude of the pressure tunnel). The mass of the water can be specified as the product of density and volume:

$$m = \rho \cdot V = \rho \cdot A_t \cdot \ell_t \tag{H5.4f}$$

where ℓ_t is the length of the pressure tunnel ($\ell_t = 13580\ m$). In turbulent flow, the inner friction of the water is proportional to the square of the velocity:

$$F_{Fr} = k \cdot A_t \cdot \rho \cdot g \cdot v_t |v_t| \tag{H5.4g}$$

where k is the friction constant ($k = 4.1\ m^{-1}\ sec^2$). These informations suffice to generate a differential equation for the tunnel velocity v_t.

At the rear end of the pressure tunnel, we can formulate the mass continuity equation (what comes in must go out), assuming that the water is ideally incompressible:

$$\dot{q}_t = \dot{q}_C + \dot{q}_p \tag{H5.4h}$$

where the mass flow rate in the pressure tunnel $\dot{q}_t$ is the product of the tunnel velocity and the tunnel cross–section:

$$\dot{q}_t = A_t \cdot v_t \tag{H5.4i}$$

the mass flow rate in the pipe is the product of the pipe velocity v_p, the pipe cross–section ($A_p = 0.6\ m^2$), and the current percentage of valve opening $S(t)$:

$$\dot{q}_p = A_p \cdot v_p \cdot S(t) \tag{H5.4j}$$

Finally, the mass flow rate into the surge tank $\dot{q}_C$ is the product of the cross–section of the surge tank and the time derivative of the water level z in the surge tank. In other words, eq(H5.4h) can be rewritten as:

$$A_t\ v_t\ dt = A_p\ v_p\ S(t)\ dt + FSW(z)\ dz \tag{H5.4k}$$

where $S(t)$ stands for either the closing characteristic or the opening characteristic of the valve which have been chosen to follow the following static characteristics (Fig.H5.4b-c):

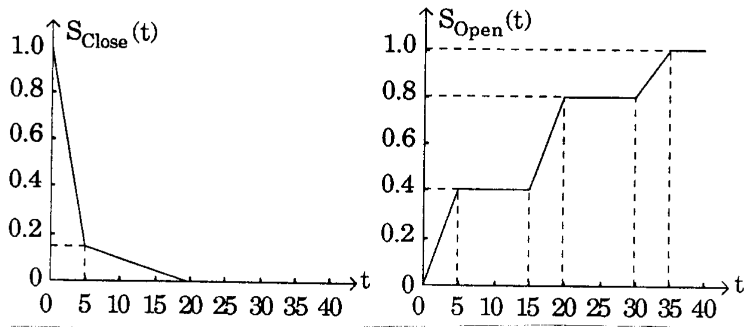

Figure H5.4b-c. Closing/opening characteristics of the valve

FWS(z) represents cross–sections of the surge tank at different altitudes. This tabular function is given in Table H5.4.

Table H5.4 Topology of the surge tank

altitude z $[m]$	surge tank surface FWS $[m^2]$
3100.0	20.0
3110.0	20.0
3120.0	100.0
3130.0	40.0
3160.0	40.0
3170.0	80.0
3180.0	80.0
3190.0	10.0
3210.0	10.0

The only quantity that we are still missing is the pipe velocity v_p. To compute this variable, we need to see what happens at the valve. Let me assume that the valve is initially open. Water flows into the turbine which carries a kinetic energy of:

$$E_k = \frac{1}{2} m v_p^2 = \frac{1}{2} \rho \; V \; v_p^2 \qquad (H5.4l)$$

The water carries also a potential energy of:

$$E_p = mgh = \rho \; V \; g \; h \qquad (H5.4m)$$

If we now start to close the valve, the total free energy doesn't change, i.e., the reduction in kinetic energy must be equal to the increase in potential energy:

$$\frac{1}{2}\rho\ \Delta V\ v_p^2 = \rho\ \Delta V\ g\ h \qquad (H5.4n)$$

and therefore:

$$v_p = \sqrt{2gh} \qquad (H5.4o)$$

Under the assumption that the change in the water level can be neglected, we can write this as:

$$v_p = \sqrt{2g(z_R - z_T)} \qquad (H5.4p)$$

The altitude of the turbine z_T is 3072 *m*.

The critical quantity that we are interested in is the *inertial pressure* at the rear end of the pressure tunnel during valve openings and valve closings. The inertial pressure can be computed as:

$$p_I = \frac{F_I}{A_t} = \frac{m\dot{v}_t}{A_t} = \frac{\rho \ell_t A_t \dot{v}_t}{A_t} = \rho \ell_t \dot{v}_t \qquad (H5.4q)$$

Simulate separately one opening and one closing of the valve over a duration of 2000 *sec* each, starting from steady-state initial conditions. Plot on separate graphs the tunnel velocity $v_t(t)$, the water level $z(t)$ in the surge tank, and the inertial pressure $p_I(t)$ as functions of time, and determine a numerical value for the maximum absolute inertial pressure.

For a configuration without a surge tank, the equations become so simple that the inertial pressure can be computed analytically. Determine the maximum value of the inertial pressure if the surge tank is being removed, and compare this value to the one found by simulation before.

[H5.5] Macros

Four bicyclists start at the coordinates indicated on Fig.H5.5. They travel with constant velocities $v_1 = 17\ km/h$, $v_2 = 14\ km/h$, $v_3 = 12\ km/h$, and $v_4 = 15\ km/h$. Each bicyclist travels at all times straight into the momentary direction of her or his next neighbor, i.e., bicyclist #1 tries to catch bicyclist #2, etc.

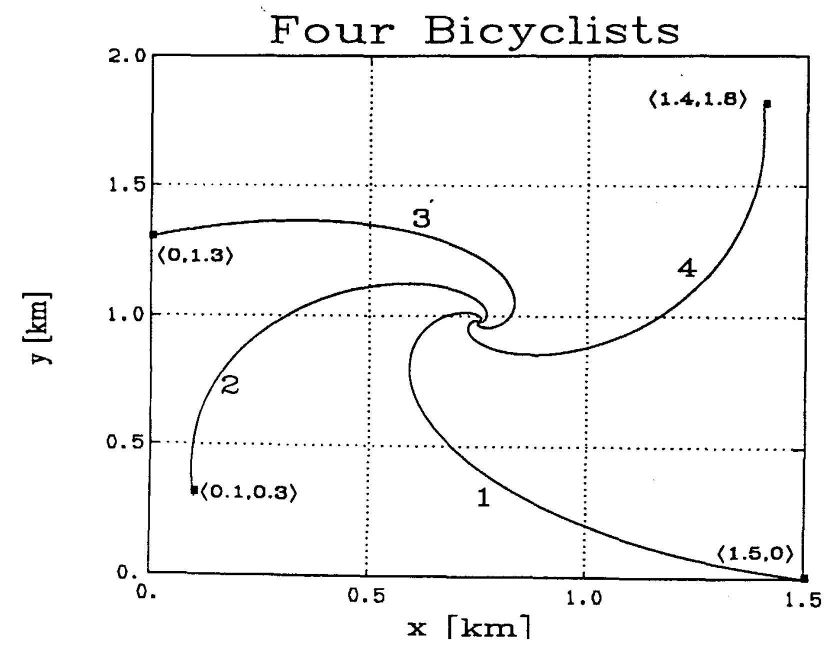

Figure H5.5. Phase plane plot of four bicyclists trying to catch each other

Compute the positional coordinates of each of the four bicyclists as functions of time. Terminate the simulation as soon as the distance between any two bicyclists has decreased to below 10 m. Develop a macro that describes the motion of any one of the four bicyclists, and call that macro four times in your simulation program.

[H5.6] Electric Power Generation

Fig.H5.6 shows the configuration of a small electric power system consisting of two synchronous generators, three transmission lines, and three different loads [5.5].

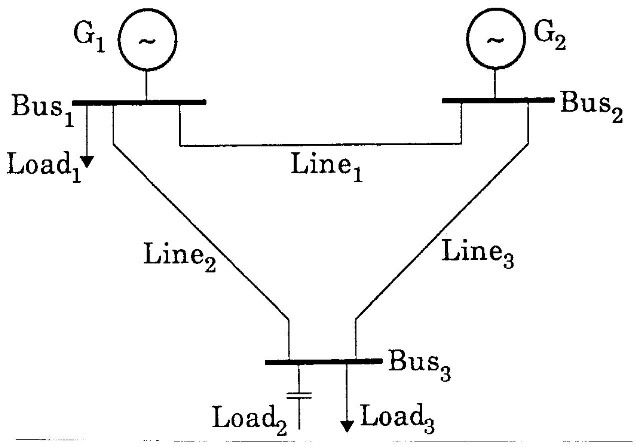

Figure H5.6. Topology of a small electric power system

It is assumed that all voltages and currents in this system are basically sinusoidal with slowly varying amplitudes and phases. We can therefore conveniently model the system with complex voltages and currents.

The generators are represented by AC voltage sources series connected with a pure reactance:

$$V_g = E_g - j\, X_d\, I_g \tag{H5.6a}$$

where E_g is the ideal voltage source, and V_g is the effective voltage that is connected to the net. I_g is the current that flows through the generator, and X_d is the (inductive) reactance of the generator. Both generators have a reactance of $X_d = 0.054\Omega$. The electric power produced by the generator is the real part of the product of the generator voltage E_g and the generator current I_g:

$$P_g = Re\{E_g \cdot I_g\} \tag{H5.6b}$$

For reasons of energy conservation, the electric power P_g must be equal to the mechanical power P_t of the water turbine that is responsible for rotating the generator. If this is temporarily not true, the phase angle between current and voltage will change to make it true. A model for this adaptation is the so-called *swing equation* [5.3]:

$$\frac{H}{\pi f_0}\ddot{\varphi} + D\dot{\varphi} = P_t - P_g \tag{H5.6c}$$

where f_0 is the net frequency (60 Hz in the United States, and 50 Hz in Europe), and H and D are generator parameters. For our two generators, we want to assume that $D_1 = D_2 = 0$, $H_1 = 30$ $Joule$, and $H_2 = 300$ $Joule$.

The transmission lines are modeled by (inductive) reactances:

$$V_{out} = V_{in} - j\ X_L\ I_L \tag{H5.6d}$$

where X_L is the reactance of the transmission line. In our example, we want to assume that all three transmission lines are equal, and have reactances of 0.05Ω.

The loads are modeled as impedances:

$$V_{Load} = Z \cdot I_{Load} \tag{H5.6e}$$

We want to analyze the reaction of the power system to various load functions. Therefore, the loads are not *a priori* specified. All we know is that the second load is purely capacitive, while the other two loads can be anything.

Create three separate DYMOLA model types for the three types of components: the generators, the transmission lines, and the loads. Then connect these components to the topology shown in Fig.H5.6. While we made the task of modeling easy and convenient, we now have to pay the price: the model does not contain enough differential equations, and therefore, the system contains several algebraic loops. Use the DYMOLA preprocessor to detect those algebraic loops, and determine how many variables are involved in each of them.

Projects

[P5.1] The Domino Game

The domino game consists of 55 stone with the dimensions $x_D = 8$ mm, $y_D = 2.4$ cm, and $z_D = 4.6$ cm. The mass of each stone is $m = 10$ g. We place these 55 stones in a series at a distance d from each other. We push the first stone, and the entire series of stones falls flat [5.2]. This is shown in Fig.P5.1a.

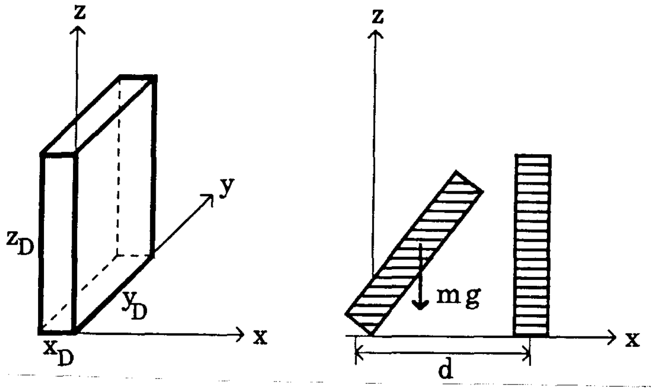

Figure P5.1a. Domino game

The question that we wish to answer is the following: At what distance d between two consecutive stones is the chain velocity v_{Ch} maximized. The chain velocity can be defined as follows:

$$v_{Ch}(k) = \frac{d}{t_{Imp}(k+1) - t_{Imp}(k)} \tag{$P5.1a$}$$

which is a discrete function. k stands for the stone $^{\#}k$. T_{Imp} is the time of impact, i.e., $t_{Imp}(k+1)$ is the time when the stone $^{\#}k$ bumps into the stone $^{\#}(k+1)$. This problem must have a non–trivial answer since, for very small values of d, the impacting stone has not yet gained a sufficiently large momentum at impact, while, for large values of d, most of the momentum is directed towards the floor already.

This is a very difficult problem to model accurately. We want to make some simplifying assumptions. First, we assume the impact to be totally *elastic*, i.e., just before impact, the impacting stone has a certain momentum $\mathcal{I} = m \cdot v$ which can be decomposed into its horizontal component $\mathcal{I}_x$ and its vertical component $\mathcal{I}_z$. At impact, the horizontal component of the momentum is passed on to the next stone, i.e., immediately after impact, the horizontal momentum of the impacting stone is zero, while the horizontal momentum of the impacted stone has taken over the momentum from the impacting stone. Second, we shall neglect the interaction between the stones after impact, i.e., the interaction is assumed to be momentary only, and thereafter, both stones are treated separately again. Each stone is simulated until it hits the ground, i.e., we shall neglect the kinematic constraints of two different stones occupying the same point in space at the same time.

At time zero, the first stone is pushed at an altitude $h = 0.75 \cdot z_D$. It thereby receives an initial horizontal momentum of $\mathcal{I} = 0.002\ kg\ m\ sec^{-1}$. This produces an initial velocity of:

$$v_0 = \frac{\mathcal{I}}{m} \qquad (P5.1b)$$

Due to Coulomb friction between the stone and the ground, the stone will not slip along the floor, but starts to roll. The initial angular velocity is therefore:

$$\omega_0 = \frac{v_0}{h} \qquad (P5.1c)$$

We can now formulate Newton's law in rotational coordinates:

$$I_y \dot{\omega} = \tau \qquad (P5.1d)$$

where I_y is the inertia of the stone to rotation around the y-axis. The inertia can be computed easily:

$$I_y = \frac{1}{3} m (x_D^2 + z_D^2) \qquad (P5.1e)$$

τ is the gravitational torque. Fig.P5.1b shows what happens while the stone falls.

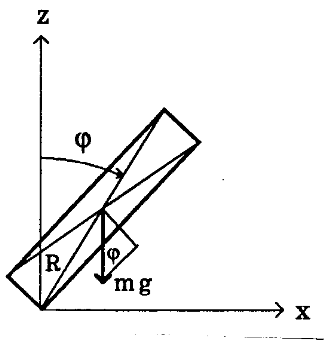

Figure P5.1b. Newton's law for the domino game

The gravitational force can be decomposed into a normal and a tangential component. Only the tangential component produces a torque which can easily be computed:

$$\tau = mg\ R\ \sin(\varphi) \qquad (P5.1f)$$

where

$$R = \frac{1}{2}\sqrt{x_D^2 + z_D^2} \qquad (P5.1g)$$

Notice that the initial angle φ_0 is negative, namely

$$\varphi_0 = -\tan^{-1}\left(\frac{x_D}{z_D}\right) \qquad (P5.1h)$$

and therefore, the initial torque is also negative. If the momentum $\mathcal{I}$ is chosen too small, the stone will never fall, but only rock for a while, and then return to its initial upright position.

Generate an ACSL macro that describes the behavior of a stone. The differential equations are multiplied by a constant c which is initially set equal to zero. When the stone is pushed (modeled using ACSL's state–event scheduling feature), it obtains its initial angular velocity ω_0, and the constant c is altered from zero to one. The stone starts moving, and a new state–event is scheduled to occur at the time of next impact. At that moment, the angular velocity of the impacting stone changes abruptly, and the stone continues to fall until $\varphi = 90°$. At that time, another state–event is scheduled which will set the constant c back to zero. We shall assume the impact with the ground to be totally *plastic*, i.e., the fallen stone does not bounce.

Call the above macro 55 times for the 55 stones of the domino game, and simulate the overall system. Plan a strategy that will optimize the chain velocity v_{Ch}. Obviously, the distance d must be chosen in the range:

$$d \in [x_d \, , \, x_D + z_D] \qquad (P5.1i)$$

ACSL permits you to branch from the *TERMINAL* section of the program (which is executed after a simulation run has been completed) back to the *INITIAL* section using a *GOTO* statement. By doing so, a next simulation run is initiated. This feature enables you to program your strategy.

Each stone is described by a second order differential equation. Consequently, the entire system order is 110 at all times, even though only a few stones move simultaneously. This is most unfortunate since this costs a lot of execution time in vain. Very few languages permit you to dynamically create/destroy processes involving differential equations, i.e., invoke a new instantiation of a macro at run time during a discrete event. One such language is COSMOS [5.12].

Try to outwit ACSL by assuming that never more than eight stones will simultaneously move at any one time. Consequently, when you push stone #9, you can simply revive the already fallen stone #1, etc. Thereby, the overall system order can be reduced from 110 to 16 which will speed up the simulation dramatically.

For the optimal distance d, plot the chain velocity over the stone number. For each simulation run, store away the last value of the chain velocity $v_{Chl} = v_{Ch}(54)$ and the currently used value of d. After all simulations have been completed, plot $v_{Chl}(d)$.

[P5.2] Robot Modeling

We wish to analyze once more the behavior of the Stanford arm (for a detailed description, *cf.* pr(P4.2)). This time, we wish to model each limb separately. Describe a general limb through a DYMOLA model type. Each two consecutive limbs are connected with a joint. Describe in another set of DYMOLA model types the different types of joints (revolute and prismatic). Build a model of the entire Stanford arm by connecting the limbs and joints together.

Employ the DYMOLA preprocessor to obtain a total set of equations describing the Stanford arm. Obviously, the kinematic constraints imposed by the connections will result in *structural singularities.* Use the differentiation algorithm outlined in this chapter to manually get rid of these singularities.

Research

[R5.1] The Modular Modeling System

The *Modular Modeling System (MMS)* [5.7] was developed by the Electric Power Research Institute (EPRI) for the simulation of various types of electric power distribution systems. Currently, two implementations of MMS exist, one in the form of a macro library for ACSL [5.16], the other as an application of Easy5 [5.1]. Both implementations suffer from the fact that the parent languages do not provide for truly modular modeling facilities. Consequently, the macros that make up for the MMS library are rather involved, somewhat clumsy, unnecessarily slow in execution, and not truly flexible. It would therefore make a lot of sense to reimplement MMS in DYMOLA which is ideally suited to provide for a flexible and totally modular parent language environment.

Reimplement MMS in DYMOLA. Since power system models can be quite large, neither SIMNON [5.4] nor DESIRE [5.13] are well suited as simulation run–time environments. Enhance DYMOLA to alternatively generate ACSL code as well. A graphical preprocessor/postprocessor for MMS exists also which was written in EASE+ [5.8]. In the context of recoding MMS in DYMOLA, I suggest to replace this preprocessor/postprocessor by HIBLIZ [5.6].

[R5.2] Structural Singularities

Design a general–purpose algorithm that reduces structural singularities by analytic differentiation. Implement this algorithm as part of the DYMOLA language. The project pr(P5.2) can serve as a test case for this algorithm.