2

Basic Principles of Continuous System Modeling

Preview

In this chapter, we shall introduce some basic concepts of continuous system modeling. By the end of this chapter, the student should be able to code simple modeling problems in some of the currently used simulation languages. The languages ACSL, DARE–P, and DESIRE are introduced in order to demonstrate the similarities that exist between the various continuous system simulation languages.

2.1 Introduction

While the continuous system simulation languages that are available on the software market today differ somewhat in terms of syntactical details, they are all derivations of the Continuous System Simulation Language (CSSL) specification generated by the CSSL committee in 1967 [2.1]. Their basic principles are thus identical. In particular, they all are based on a *state–space description* of the system equations, i.e., on a set of first order ordinary differential equations (ODE's) of the form:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{2.1}$$

A typical simulation language may allow us to express a representative model through the set of equations:

```
DYNAMIC
  thrust = f(t)
  h.     = v
  v.     = a
  a      = (1/m) * (thrust - m * g)
  m.     = -c1*abs(thrust)
  g      = c2/(h + r) * *2
END
```

which describes the vertical motion of a rocket that is just about to perform a soft landing on the surface of the moon. This model contains three *state variables*, namely the altitude h, the vertical component of the velocity v, and the mass m of the rocket describing a *third order model* of the rocket dynamics. The dot (.) denotes the first derivative with respect to the independent variable which, in simulation languages, is traditionally assumed to be *time* (t). This model also specifies three additional *auxiliary variables*, namely the vertical component of the rocket's acceleration a which encodes Newton's law applied to the rocket's rigid body, the gravity force g which increases quadratically with decreasing altitude of the rocket, and the *thrust* which is specified as an externally computed function of the simulation clock t. The model also references three constants, namely the lunar radius r, the gravitational constant $c2$, and the fuel efficiency constant $c1$. The first two *state equations* (for the state variables h and v) denote the mechanical interrelation between position, velocity, and acceleration of a rigid body, while the third state equation (for the state variable m) describes the reduction of the rocket's mass due to fuel consumption which is assumed to be proportional to the absolute value of the applied thrust.

This simple model teaches us a number of things. First, we notice that some of the variables, such as a, are referenced *before* they have been defined. In most programming environments, this would result in a run time exception, but not so in CSSL's. The underlying physical phenomena which are captured through the equations of the simulation model are all taking place *in parallel*. This is reflected in CSSL's by allowing the user to specify his equations as *parallel code*. The sequence of the equations that make up the dynamic model of the system is entirely immaterial. An *equation sorter* which is an intrinsic part of most CSSL's will sort the equations at compilation time into an executable sequence.

An immediate consequence of the above decision is the rule that no variable can be defined more than once (since otherwise, the sorter wouldn't know which definition to use). CSSL's belong to the class of

single assignment languages (SAL's). All time dependent variables (state variables and auxiliary variables) must be defined *exactly once* inside the dynamic model description section of the simulation program.

The dynamic model equations are really of a *declarative* rather than an *executive* nature. In CSSL's, the equal sign (=) denotes *equality* rather than *assignment*. (Some programming languages, such as PASCAL, distinguish between these two different meanings of the equal operator by using a separate operator symbol, namely "=" to denote *equality* and ":=" to denote *assignment*. However, other programming languages, such as FORTRAN, don't make this distinction which is bound to create a certain degree of confusion.) A statement such as:

$$i = i + 1 \tag{2.2}$$

which is one of the most common statements in traditional programming languages, meaning that the value of the integer variable i is to be incremented by one, is entirely meaningless in a CSSL type language. If we interpret eq(2.2) as a *mathematical formula*, the formula is simply incorrect since we then can cancel the variable i from both sides of the equal sign which now implies that 0 is equal to 1 which is obviously not true. On the other hand, if we interpret eq(2.2) as a *declarative statement* (which reflects a little more accurately what the simulation languages do) then we realize that we are confronted here with a *recursive declaration* which makes the sorting algorithm as helpless as I am when I read in one of my *TIME-LIFE* cookbooks that the recipe for *taratoor* requires one table spoon of *tahini* while the recipe for *tahini* calls for one table spoon of *taratoor* (I meanwhile solved that problem — I buy *tahini* in the store).

2.2 The Algebraic Loop Problem

The above discussion unveils that not all problems related to equation sorting are solved by simply requesting that every variable be declared exactly once. The above example can be expressed in terms of a CSSL notation as:

$$taratoor = f(tahini) \tag{2.3a}$$
$$tahini = g(taratoor) \tag{2.3b}$$

With this program segment, the sorter gets stuck exactly the same way that I did. The typical response of most CSSL's in this case will be to flag the program as non–executable, and return an error message of the type: *"Algebraic loop detected involving variables taratoor and tahini"*. Arbitrarily many variables can be involved in an *algebraic loop*.

2.3 Memory Functions

Let us look a little closer at some of the equations of our rocket model, for example:

$$vdot = a \tag{2.4a}$$
$$a = (1/m) * (thrust - m * g) \tag{2.4b}$$
$$v = \mathrm{INTEG}(vdot, v0) \tag{2.4c}$$

I used here a slightly different notation which is also commonly found in many CSSL's. Instead of specifying the model in a state–space representation (with the integration operation being implied), some CSSL's provide for an explicit integration operator (called *INTEG* in our example). This notation is a little more bulky, but it has the advantage that the initial condition v_0 can be specified as part of the dynamic model description rather than being treated separately together with the constants.

From this description, we could get the impression that the variable v is a function of the variable *vdot* (through eq(2.4c)), that *vdot* in turn is a function of the variable a (through eq(2.4a)), and that finally a is a function of v (through eq(2.4b)).

Did we just detect an *algebraic loop* involving the three variables v, *vdot*, and a? To answer this question, we need to look a little more closely into the process of *numerical integration*. As was mentioned already in Chapter 1, it is necessary to discretize the continuous process of numerical integration to make it treatable on a digital computer. Many schemes exist that demonstrate how this can be accomplished, and we shall discuss those intimately in the second volume of this text. However, for now, let us just look at the two simplest schemes that can be devised. Fig.2.1 depicts the process of numerical integration.

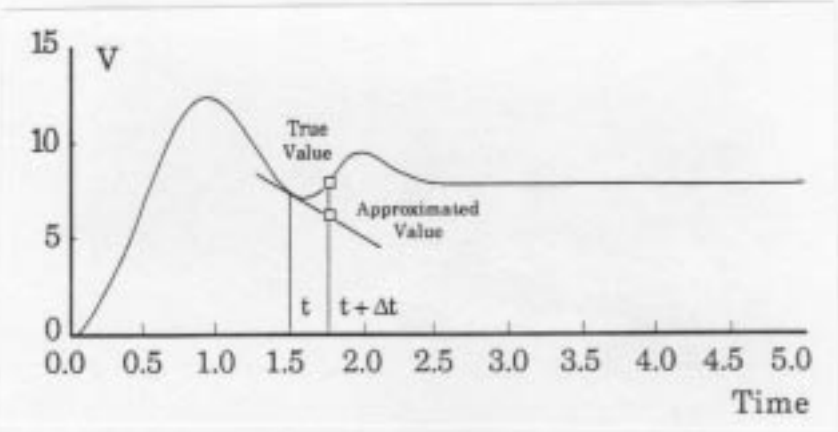

Figure 2.1. A simple scheme for numerical integration.

Given the value of our state variable v at any time t, we can approximate the value of v some Δt time units later through the formula:

$$v(t + \Delta t) \approx v(t) + \Delta t * \dot{v}(t) \tag{2.5}$$

which is commonly referred to as *Euler's integration rule.*

Since v at time $t = t_0$ is given as v_0, we can immediately evaluate eq(2.4b), followed by eq(2.4a). At this point, we have evaluated $\dot{v}(t_0)$. Therefore, we can now evaluate eq(2.5) to find $v(t_0 + \Delta t)$. Now, we can again evaluate eq(2.4b), followed by eq(2.4a) to find $\dot{v}(t_0 + \Delta t)$, a.s.f.

Obviously, the integration function has broken the algebraic loop *since it depends on values of variables at past values of time only.* A function which has this property is called a *memory function.* Any memory function will break algebraic loops, and thus, the sorter must know whether or not a function that it comes across is a memory function. The two most prominent memory functions in continuous system simulation are the *integration function* and the *delay function.* Some CSSL's allow the user to declare his or her own additional memory functions.

2.4 Explicit Versus Implicit Integration

Let us repeat the previous discussion with a slightly modified integration scheme. Fig.2.2 demonstrates this scheme.

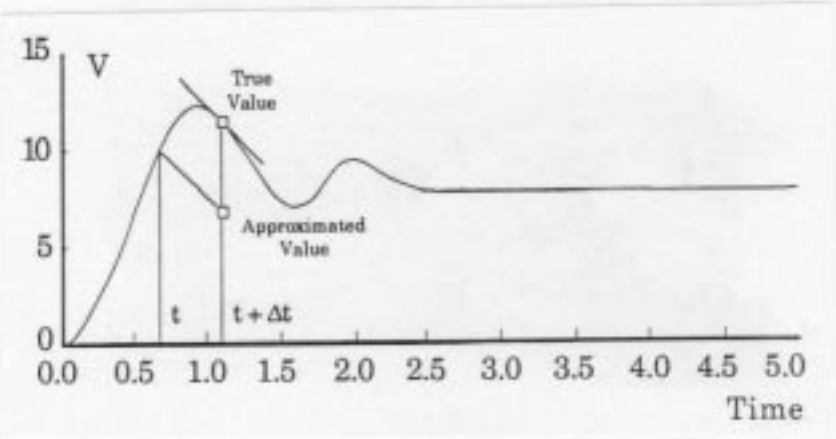

Figure 2.2. Numerical Integration by Backward Euler Technique.

In this scheme, the solution $v(t + \Delta t)$ is approximated using the values of $v(t)$ and $\dot{v}(t + \Delta t)$ through the formula:

$$v(t + \Delta t) \approx v(t) + \Delta t * \dot{v}(t + \Delta t) \tag{2.6}$$

This scheme is commonly referred to as the Backward Euler integration rule.

As can be seen, this integration formula depends on current as well as past values of variables. It is thus *not* a memory function. Consequently, it will not break up the algebraic loop. It can easily be verified that, in order to compute $\dot{v}(t_0 + \Delta t)$, we require knowledge of $a(t_0 + \Delta t)$ (according to eq(2.4a)) which in turn requires knowledge of $v(t_0 + \Delta t)$ (according to eq(2.4b)). However, in order to compute $v(t_0 + \Delta t)$ (according to eq(2.6)), we need to know $\dot{v}(t_0 + \Delta t)$. In other words, the algebraic loop has not been broken.

Integration algorithms that are described by memory functions are therefore often referred to as *explicit integration techniques*, whereas

algorithms which are not of the memory function type are called *implicit integration techniques*. Although the implicit integration techniques are advantageous from a numerical point of view (as we shall learn later), the additional computational load created by the necessity to solve simultaneously a set of non–linear algebraic equations at least once every integration step makes them undesirable for use in CSSL's. Therefore, all integration techniques that are commonly available in CSSL type languages are of the explicit type.

2.5 Implicit Loop Solvers

Let us return once more to the algebraic loop that we had met previously in this chapter. Clearly the best way to deal with algebraic loops is to solve them *manually* and replace the "delinquent" equations by a set of new equations which are now explicitly solvable.

Let us assume that the sorter found a set of two algebraically coupled equations:

$$y = -2 * x + 3 * u \tag{2.7a}$$
$$x = -y/3 + 7 * u \tag{2.7b}$$

We can easily satisfy the sorter's needs by manually solving these two equations as follows:

$$y = -33 * u \tag{2.8a}$$
$$x = +18 * u \tag{2.8b}$$

which can now be inserted in place of the two original equations.

However, this technique does not always work. Often, the set of equations does not have an analytical solution. In that case, we would like to be able to "buy tahini in the store". Some CSSL's, such as CSMP–III [2.4], provide a special mechanism for this purpose which is called an *implicit loop solver*. CSMP–III enables us to reformulate the problem as follows:

```
tahini      = IMPL(store_tahini, errmax, next_tahini)
taratoor    = f(tahini)
next_tahini = g(taratoor)
```

which is a (somewhat clumsy) way to specify the following algorithm. First, we buy a can of *tahini* in the store (*store_tahini*). Then, we produce one batch of *taratoor*. However, it could well be that the *store_tahini* is somewhat different from our favorite *tahini* recipe. Therefore, we now use the just produced *taratoor* to make a new version of *tahini* (called *next_tahini*), from which we then can produce a new batch of *taratoor*, a.s.f., until two consecutive batches of *tahini* taste almost the same (their difference is smaller than *errmax*). (The third parameter of the *IMPL* function provides the compiler with the name of the assignment variable of the *last* statement that belongs to the algebraic loop.)

Now, remember that this iteration must take place once per function evaluation, i.e., at least once every integration step. In other words, the next day, we have to go and buy a new can of *store_tahini*, and the process starts all over again. After a short while, we may decide that this algorithm is wasteful. Surely, the final solution of last day's *tahini* is a better starting value for today's *tahini* than *store_tahini*. In other words, I should save one table spoon of last day's *tahini*, and put it in the freezer for reuse today rather than starting from scratch with a new can of *store_tahini*.

CSMP–III allows us to formulate this modified algorithm as follows:

```
INITIAL
  last_tahini = store_tahini
DYNAMIC
  ...
  tahini      = IMPL(last_tahini, errmax, next_tahini)
  taratoor    = f(tahini)
  next_tahini= g(taratoor)
  ...
  NOSORT
  last_tahini = tahini
END
```

2.6 Procedural Sections

Many CSSL's don't provide for an *IMPL* function. However, other mechanisms are available that can replace this construct easily. One

such mechanism is the *procedural section* which is offered by almost all CSSL type languages.

A procedural section can be considered a "sandwich" equation, i.e. a set of regular *procedural statements* that are treated by the sorter like one equation to be sorted as a whole with all the other equations, while the assignment statements which form the procedural block stay together and are left unchanged. The header of the procedural section instructs the equation sorter about the place where the section needs to be inserted. The sorter *never* checks the inside of the procedural section itself.

For instance, the previous algorithm could be expressed in DARE–P [2.9] as follows:

```
$D1
   ...
   PROCED tahini = last_tahini
       dish_count = 0
10     taratoor   = f(last_tahini)
       tahini     = g(taratoor)
       dish_count = dish_count + 1
       IF (dish_count.gt.max_dish) GO TO 20
       IF (abs(tahini - last_tahini).gt.errmax) GO TO 10
       GO TO 30
20     WRITE(6,21)
21     FORMAT(' Iteration failed to converge')
30     CONTINUE
   ENDPRO
   ...
   PROCED dummy = tahini
       last_tahini = tahini
   ENDPRO
END
last_tahini = store_tahini
END
```

As can be seen, DARE–P employs FORTRAN statements to describe procedural sections. This is understandable since DARE–P is compiled (preprocessed) into FORTRAN. Most CSSL's use their *intermediate language* to express procedural sections.

Note that DARE–P "procedures" serve a completely different purpose than procedures in most of the other programming languages

(such as PASCAL). They are used as structuring elements to declare "sandwich" equations. The outputs of the procedure are specified to the left of the equal sign (separated by comma), while its inputs are specified to the right of the equal sign. The sorter will place the procedural block such that all its inputs have been evaluated before the procedure is computed, and that none of its outputs are used before the procedure has been computed. In other words, the inputs and outputs of the procedure follow the SAL rule, while this is not true for assignments inside the procedure.

In our example, the first procedure is used to program out the iteration loop which was implied by the previous *IMPL* construct. The iteration loop ends when we can't tell subsequent batches of *tahini* apart any longer, or when all dishes are dirty, which ever occurs first. The second procedure is used to break the algebraic loop. Since the sorter never looks inside the procedure, it will not detect that *last_tahini* is actually being redefined inside the second procedure. However, the header information of the two procedures will ensure that the second procedure is placed *after* the first since it needs *tahini* as an input, whereas the same variable *tahini* was declared as an output of the first procedure. Each *PROCED* is accompanied by an *END* statement to mark the end of the procedure.

The $D1 statement marks the beginning of the dynamic model description. The second to last *END* statement marks the end of the dynamic model description. Between this *END* and the final *END* of the code segment, DARE–P expects the declaration of constants and initial conditions.

2.7 The Basic Syntax of Current CSSL's

Let me formulate the simple lunar landing problem in terms of three current CSSL's, namely ACSL [2.7], DARE–P [2.9], and DESIRE [2.5]. This is to demonstrate how similar the various CSSL's are in their basic formalisms, i.e., to show that, once we understood one, we really know them all.

In ACSL [2.7], the problem can be formulated as follows:

```
PROGRAM Lunar Landing Maneuver
  INITIAL
    constant ...
      r = 1738.0E3,  c2 = 4.925E12,  f1 = 36350.0, ...
      f2 = 1308.0,  c11 = 0.000277,  c12 = 0.000277, ...
      h0 = 59404.0,  v0 = -2003.0,  m0 = 1038.358, ...
      tmx = 230.0,  tdec = 43.2,  tend = 210.0
    cinterval cint = 0.2
  END $ "of INITIAL"
  DYNAMIC
    DERIVATIVE
      thrust = (1.0- step(tend)) * (f1 - (f1 - f2)*step(tdec))
      c1     = (1.0- step(tend)) * (c11 - (c11 - c12)*step(tdec))
      h      = integ(v, h0)
      v      = integ(a, v0)
      a      = (1.0/m) * (thrust - m * g)
      m      = integ(mdot, m0)
      mdot   = -c1*abs(thrust)
      g      = c2/(h + r) * *2
    END $ "of DERIVATIVE"
    termt (t.ge.tmx .or. h.le.0.0  .or. v.gt.0.0)
  END $ "of DYNAMIC"
END $ "of PROGRAM"
```

From the previous discussions, this program should be almost self-explanatory. *Step* is an ACSL system function the output of which is zero as long as the system variable t (the simulation clock) is smaller than the parameter (in our case *tdec* and *tend*). Therefore, the *thrust* takes a value of $f1$ for t smaller than *tdec*. It then takes a value of $f2$ for t between *tdec* and *tend*, and it takes a value of 0.0 thereafter. The fuel efficiency constant $c1$ is computed by the same mechanism. This equation was introduced since the main retro motor which produces the thrust $f1$ and the three vernier engines which together produce the thrust $f2$ may have a different fuel efficiency. *Tmx* denotes the final time of the simulation. *Cint* denotes the *communication interval*, i.e., it tells ACSL how often results are to be stored in the simulation data base. Finally, *termt* is a *dynamic termination criterion*. Whenever the logical expression of the *termt* statement becomes true, the simulation will terminate. Notice that *tmx* is not a system constant, and must be manually tested in the *termt* statement.

Notice that this program does not contain any *output statements*. In the compilation process, the ACSL code is first preprocessed into

FORTRAN by the *ACSL preprocessor*, then it is compiled further into machine code, and thereafter it is linked with the *ACSL run-time library*. During execution of the resulting code, ACSL automatically switches to an *interactive mode* in which parameter values can be modified, simulation runs can be performed, and simulation outputs can be plotted.

The two-step compilation is standard practice in most CSSL's since it provides *machine independence*, as it can be assumed that all computers provide for a FORTRAN compiler.

Now, let me write down the corresponding DARE-P [2.9] code:

```
$D1
   THRUST = (1.0 - STPE) * (F1 - (F1 - F2) * STPD)
   C1     = (1.0 - STPE) * (C11 - (C11 - C12) * STPD)
   H.     = V
   V.     = A
   A      = (1.0/XM) * (THRUST - XM * G)
   XM.    = -C1*ABS(THRUST)
   G      = C2/(H + R) **2
   STPE   = STP(T,TEND)
   STPD   = STP(T,TDEC)
   TERMINATE -H * V
$F
   FUNCTION STP(T,TON)
   STP = 0.0
   IF (T.GE.TON) STP = 1.0
   RETURN
   END
END
R = 1738.0E3,  C2 = 4.925E12,  F1 = 36350.0
F2 = 1308.0,  C11 = 0.000277,  C12 = 0.000277
H = 59404.0,  V = -2003.0,  XM = 1038.358
TMAX = 230.0,  NPOINT = 301,  TDEC = 43.2,  TEND = 210.0
END
*LUNAR LANDING MANEUVER
GRAPH H
GRAPH V
END
```

DARE-P is a much older (and a much more old-fashioned) language than ACSL. It was originally designed for CDC machines which did not operate on a full ASCII character set, and consequently, even today, many versions of DARE-P don't support lower case characters. The mass m had to be renamed into XM since any variable starting with I, J, K, L, M, or N is considered to be of type *INTEGER*,

and no mechanism exists in DARE–P to reassign the type of such variables. In the previous DARE–P program segment, I had ignored these restrictions in order to improve the readability of the code, and I shall do so again in the future since, contrary to many other areas of life, in programming languages, age does not deserve reverence. Nevertheless, DARE–P still has its beauties and advantages as we shall see later. One of its true advantages is the fact that its syntax is *very* easy to learn.

Other than that, the two programs are similar. DARE–P prefers the "dot"–notation over the *INTEG* operator. The dynamic termination condition is here called *TERMINATE* and operates on a numerical rather than a logical expression. The simulation run terminates whenever the numerical expression associated with the *TERMINATE* statement becomes negative. This is a little less powerful than ACSL's technique since it could eventually happen that both the altitude h and the velocity v change their sign within the same integration step, and under those circumstances, the *TERMINATE* condition would not trigger. DARE–P does not allow the user to specify several termination conditions on separate *TERMINATE* statements. *STP* is not a DARE–P system function, but can easily be created by use of the $F block in which arbitrary FORTRAN subprograms can be coded. (ACSL provides for the same mechanism by permitting the user to code her or his FORTRAN subprograms following the final *END* statement of the ACSL program.) The final time is here called *TMAX* rather than *tmx*, and is treated as a system constant which is automatically tested, and the communication interval is determined indirectly by specifying how many data points are to be stored during the simulation run. $NPOINT = 301$ does not give us a communication interval of 0.2 *sec*, but it is the largest number allowed in DARE–P before some internal arrays overflow. (FORTRAN programming is so much fun!)

The model description is followed by a program section in which constants and initial conditions are specified. The final code section of any DARE–P program lists the output commands.

Notice that the segment separators in DARE–P are *column sensitive*. All $ block markers ($D1 and $F in our example) must be coded with the $ starting in column 2, and the three DARE–P *END* separators (before and after the constants, and following the output declarations) must be placed in columns 1 to 3. (The DARE–P syntax is an excellent example of how *not* to design a language grammar, however in the early 1970's when DARE–P was developed, software engineering was still in its infancy, and as a university product,

DARE–P did not carry the financial means behind that would have permitted a constant upgrading of the product.)

Let us finally look at DESIRE [2.5]. The following DESIRE program is equivalent to the two previous programs.

```
-----------------------------------
-- CONTINUOUS SYSTEM
-- Lunar Landing Maneuver
-----------------------------------
-- Constants
r = 1738.0E+3  |  c2 = 4.925E+12  |  f1 = 36350.0
f2 = 1308.0  |  c11 = 0.000277  |  c12 = 0.000277
tdec = 43.2  |  tend = 210.0
-- Initial conditions
h = 59404.0  |  v = -2003.0  |  m = 1038.358
-----------------------------------
TMAX = 230.0  |  DT = 0.1  |  NN = 1151
-----------------------------------
-- auto scale
scale = 1
XCCC = 1
label TRY
drunr
if XCCC < 0 then XCCC = -XCCC
                 scale = 2 * scale
                 go to TRY
            else proceed
-----------------------------------
DYNAMIC
-----------------------------------
thrust  = ((f1 - f2)*swtch(tdec - t) + f2)*swtch(tend - t)
c1      = ((c11 - c12)*swtch(tdec - t) + c12)*swtch(tend - t)
mdot    = -c1*abs(thrust)
g       = c2/(h + r) ^ 2
a       = (1.0/m) * (thrust - m * g)
d/dt h  = v
d/dt v  = a
d/dt m  = mdot
-----------------------------------
OUT
term -h
term v
hs = h * 1E-5  |  ms = m * 5E-4  |  vs = v * 5E-4
dispt hs, vs, ms
-----------------------------------
/--
/PIC 'lunar.prc'
/--
```

While ACSL was designed for moderately sized industrial simulation problems, and DARE–P was designed for easy learning (class room environments), DESIRE has been designed for maximum interactivity, and for ultra–fast execution speed.

These goals have considerably influenced the software design. DESIRE is one of the few CSSL's that does not base upon a target language for improved portability since portability goes always at the expense of execution speed. The statements above the *DYNAMIC* declaration describe the *experiment* to be performed on the model, i.e., they constitute procedural code to be executed only once. These statements are coded in an *interpreted* enhanced BASIC, and are therefore slow in execution. However, since this code is executed only once rather than constantly during the simulation run, speed is not the issue. The statements following the *DYNAMIC* declaration describe the *dynamic model*. They are *micro–compiled* into *threaded code* when the simulation starts. A simulation run is performed whenever, during execution of the experiment section, a *drun* or *drunr* statement is met. Although the model description code is heavily optimized, the execution of the micro–compiler is ultra–fast. The above program compiles within less than 0.1 *sec*, and thereafter, the program is ready to run. However, in order to keep the compilation time down, it was decided *not* to provide for an automated equation sorter, i.e., it is the user's responsibility to place the equations in an executable sequence.

DESIRE is modeled after the *analog computers* of the past. It provides for the ultimate in flexibility, interactivity, and responsiveness. The *dispt* statement, for instance, invokes a *run–time display*, i.e., the user can view the results of his or her simulation run as they develop. The price to be paid for this sportscar among the simulation languages is a certain reduction in programming comfort, program reliability (compile–time checks), and program robustness (run–time checks).

DESIRE is by far the most *modern* of the three languages. While a version exists that executes on VAX/VMS, the beauty of DESIRE lies really in its PC implementation. The execution speed of programs that consist of several hundred differential equations on a 386 class machine is breath–taking. This is due to the fact that the user doesn't share this powerful machine with anybody else. As the designer of the software, Granino Korn, always says: "Time–sharing is for the birds". DESIRE certainly goes with the trend of modern design software which is away from the main frames onto ever more

powerful engineering workstations such as the 386's, the MacIntosh–II, and the MicroVAX and SUN workstations.

Other than that, the program listed above requires little additional explanation. DESIRE is *case sensitive*, i.e., *TRY*, *Try*, and *try* are three perfectly good variable names, but they denote three *different* variables. This is important to remember in particular in the context of system variables (for example, the simulation clock is called t and not T). *NN* is a system variable that serves the same purpose as *NPOINT* in DARE–P. The integration step size *DT* must be specified since the default integration algorithm of DESIRE is a *fixed–step Heun algorithm* rather than the *variable–step* 4^{th} *order Runge–Kutta algorithm* which is most commonly used as the default algorithm of CSSL's. *XCCC* (what a name!) is a system variable which is set negative by DESIRE when a run–time display variable exceeds the scaling bounds (defined by the system variable *scale*). Since DESIRE simulation runs are so fast, we can afford to simply repeat the entire simulation with a larger scale for the display, rather than bother to store the previous data away and recover them from memory. *swtch* is a DESIRE system function which uses a slightly different syntax than ACSL's *step* function, but performs the same task. DESIRE uses a "dot"–notation (as DARE–P), but a different syntax (d/dt).

DESIRE's *OUT* block is similar to ACSL's portion of the *DYNAMIC* block which is outside the *DERIVATIVE* section. Statements in the *OUT* block are executed only once per communication interval rather than during every integration step. This is not important except for speeding up the execution of the simulation run. *hs*, *ms*, and *vs* are scaled variables. Scaling was necessary to make all three variables comparable in amplitude, since DESIRE's run–time display uses the same scaling factor for all dependent variables (again in order to speed up the execution). The *term* statement operates on a numerical condition (as in DARE–P), but this time, it triggers whenever the associated expression becomes *positive*. Moreover, DESIRE tolerates several separate *term* statements to be specified in a row.

Let us now discuss the results of this simulation study. Fig.2.3 shows the time trajectories of five of the simulation variables.

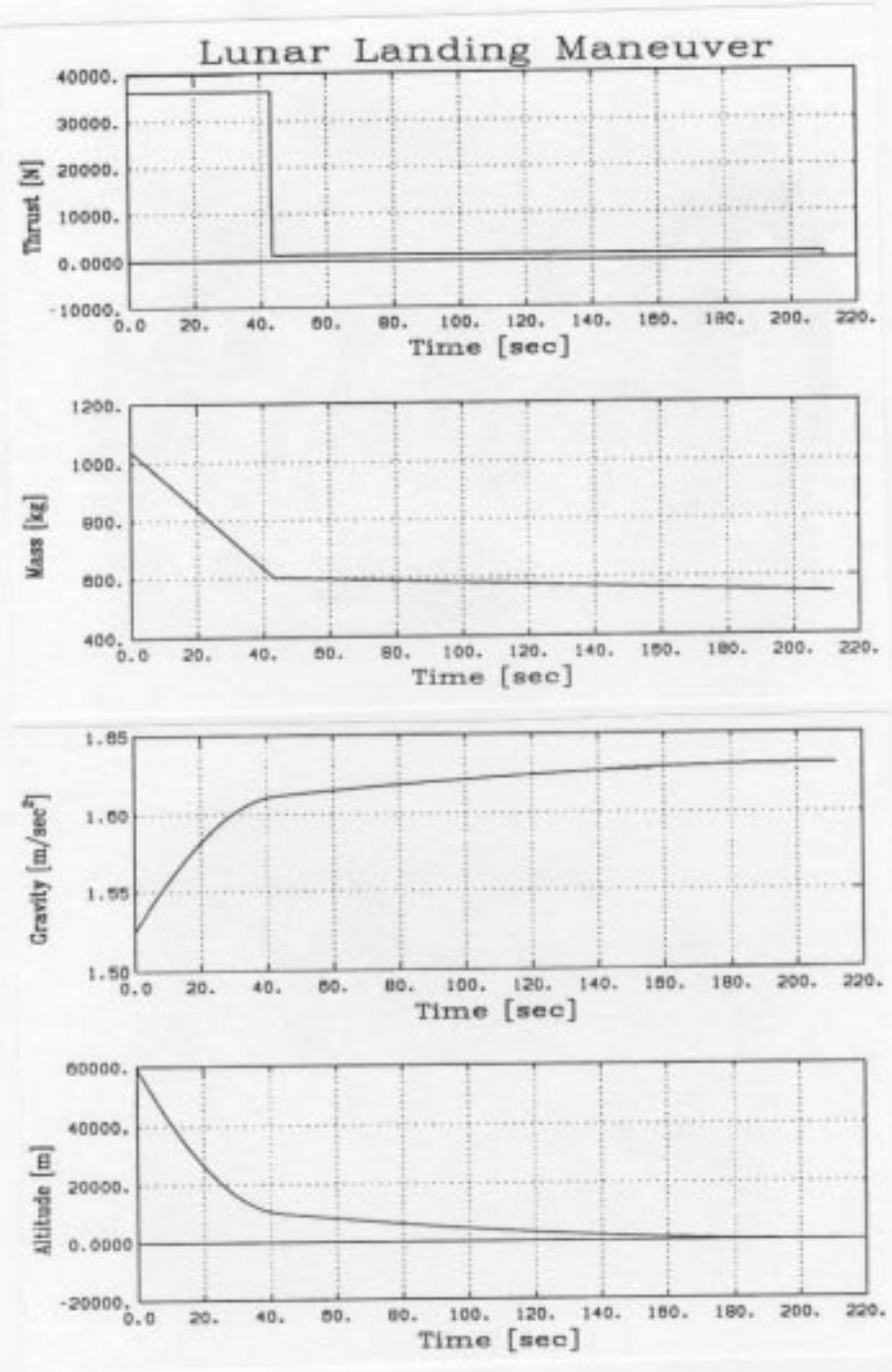

Figure 2.3. Results from the Lunar Landing Maneuver.

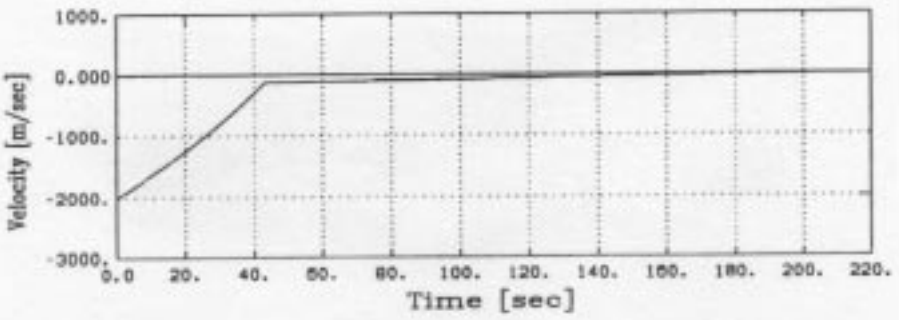

Figure 2.3. Results from the Lunar Landing Maneuver (continued).

The main retro motor is fired at time zero, and remains active during the first 43 *sec* of the landing maneuver slowing the landing module down from an initial vertical velocity of 1800 m/sec to about 150 m/sec. From then on, only the three vernier engines are used. At time 210 *sec*, the module has landed. (A soft landing requires a vertical velocity of less than 5 m/sec at impact.) During these 210 *sec*, the module has lost half of its initial weight due to fuel consumption.

The results shown on these graphs were produced by the ACSL program which, however, was executed under control of yet another program, called CTRL–C [2.8], a technique that we shall use frequently in this textbook. The graphs themselves were produced after the simulation run was completed by use of CTRL–C's graphic routines. A similar interface exists also between ACSL and Pro–MATLAB [2.6]. More and other simulation systems will be discussed at a later time.

The main purpose of this section was to illustrate how similar the various CSSL's are to each other. Once we have mastered any one of them, we basically know them all. In this textbook, we shall mostly use ACSL for demonstration purposes since we believe it to be the most flexible, the most convenient, and the most widely used among the currently available CSSL's. However, for those who have another CSSL software installed in their computer, it should not prove overly difficult to transcribe the presented concepts in terms of their CSSL software.

2.8 Discontinuity Handling

Already the first simple example of a continuous system which we presented in this chapter showed the need to model *discontinuous systems*. During the course of the simulation, the thrust of the rocket changed abruptly twice. This was not an accident. Discrete changes within continuous systems are extremely common, and any decent CSSL should offer means to code such discrete changes in a convenient and efficient way. Unfortunately, this is not the case. The need for such a language element had not been anticipated by the CSSL committee [2.1], and therefore, none of the subsequently developed CSSL systems offered such a feature. This is the drawback of too early a software standardization — it tends to freeze the state–of–the–art, and thereby hampers future developments. This issue had not been addressed to its full extent until 1979 [2.2,2.3]. Among the three systems presented in this chapter, only ACSL offers a decent discontinuity handling facility.

Some discrete changes are known in advance and can be *scheduled* to occur at a particular point in time. This type of discontinuity is called a *time–event*. The change in the rocket thrust belongs to this type of discontinuity. Time–events can be scheduled during the execution of the *INITIAL* block. At the time of occurrence, the *DERIVATIVE* block will be interrupted, and a special *DISCRETE* block is executed once. During the execution of the *DISCRETE* block, new time–events can be scheduled to happen at an arbitrary time in the future.

Let me illustrate this concept by means of the previous example. A more elegant reformulation of the lunar landing maneuver in ACSL is given below. This version of the code contains two time–events, one to shut down the main retro motor, and the other to shut down the vernier engines. The time of shut–down has been precomputed, and the two events can therefore be scheduled from within the *INITIAL* block.

```
PROGRAM Lunar Landing Maneuver
  INITIAL
    constant ...
      r = 1738.0E3,  c2 = 4.925E12,  f1 = 36350.0, ...
      f2 = 1308.0,  c11 = 0.000277,  c12 = 0.000277, ...
      h0 = 59404.0,  v0 = -2003.0,  m0 = 1038.358, ...
      tmx = 230.0,  tdec = 43.2,  tend = 210.0
    cinterval cint = 0.2
    schedule shutlg .at. tdec
    schedule shutsm .at. tend
    ff = f1
    cc = c11
  END $ "of INITIAL"
  DYNAMIC
    DERIVATIVE
      thrust = ff
      c1     = cc
      h      = integ(v, h0)
      v      = integ(a, v0)
      a      = (1.0/m) * (thrust - m * g)
      m      = integ(mdot, m0)
      mdot   = -c1*abs(thrust)
      g      = c2/(h + r) * *2
    END $ "of DERIVATIVE"
    DISCRETE shutlg
      ff = f2
      cc = c12
    END $ "of DISCRETE shutlg"
    DISCRETE shutsm
      ff = 0.0
      cc = 0.0
    END $ "of DISCRETE shutsm"
    termt (t.ge.tmx .or. h.le.0.0  .or. v.gt.0.0)
  END $ "of DYNAMIC"
END $ "of PROGRAM"
```

At time *tdec*, the retro motor is shut down. This is being accomplished by the time–event *shutlg*. At time *tend*, also the vernier engines are shut down. This is being accomplished by the time–event *shutsm*. Notice that the two variables *ff* and *cc* must be assigned initial values. This solution is much more readable than the previous one. It also has numerical benefits as we shall see in the second volume of this textbook.

Sometimes, time–events recur in constant time intervals. Although such events could be coded in exactly the same way by simply scheduling the next occurrence of the same event from within the *DISCRETE* block itself, this case is so common that ACSL offers a

simplified way of scheduling recurring events. If a *DISCRETE* block contains an *INTERVAL* statement, it is *implicitly scheduled* to occur once every sampling interval. No *schedule* statement is necessary in that case. The following program shows how the set of difference equations:

$$x_1(k+1) = (1-a)x_1(k) + b\ x_1(k)x_2(k) \tag{2.9a}$$
$$x_2(k+1) = (1+c)x_2(k) - d\ x_1(k)x_2(k) \tag{2.9b}$$

can be coded in ACSL.

```
PROGRAM Difference Equations
  INITIAL
    constant ...
      a = 0.07,  b = 0.02,  c = 0.05,  d = 0.02, ...
      tmx = 100.0
    cinterval cint = 1.0
    x1 = 2.5
    x2 = 1.7
  END $ "of INITIAL"
  DYNAMIC
    DISCRETE difrnc
      interval Ts = 1.0
      PROCEDURAL (x1,x2 =)
        x12 = x1 * x2
        x1new = (1.0 - a) * x1 + b * x12
        x2new = (1.0 + c) * x2 - d * x12
        x1 = x1new
        x2 = x2new
      END $ "of PROCEDURAL"
    END $ "of DISCRETE difrnc"
    termt (t.ge.tmx)
  END $ "of DYNAMIC"
END $ "of PROGRAM"
```

The *DISCRETE* block *difrnc* is implicitly scheduled to occur once every *Ts* time units. The communication interval *cint* is chosen to be identical with the sampling interval. The model is executed over 100 iterations of the set of difference equations. The *PROCEDURAL* block is ACSL's equivalent of DARE–P's *PROCED* block which we met before. It is needed here to prevent the equation sorter from detecting algebraic loops between *x1* and *x1new*, and between *x2* and *x2new*.

Sometimes, discontinuities depend on the model states rather than on time. Such discontinuities are therefore called *state–events*. For

example, we could reformulate our lunar lander example such that the main retro motor is shut down when the landing module has reached an altitude of 9934 m rather than at time 43.2 sec. This may, in fact, be a more robust formulation of our problem since it may be less sensitive to disturbances such as a somewhat inaccurately guessed initial mass value.

State–events cannot be scheduled ahead of time. ACSL "schedules" state–events from within the *DERIVATIVE* section of the model whenever a specified *state condition* is met. The following code shows the reformulated lunar landing maneuver.

```
PROGRAM Lunar Landing Maneuver
  INITIAL
    constant ...
      r = 1738.0E3,  c2 = 4.925E12,  f1 = 36350.0, ...
      f2 = 1308.0,  c11 = 0.000277,  c12 = 0.000277, ...
      h0 = 59404.0,  v0 = -2003.0,  m0 = 1038.358, ...
      tmx = 230.0
    cinterval cint = 0.2
    ff = f1
    cc = c11
  END $ "of INITIAL"
  DYNAMIC
    DERIVATIVE
      thrust = ff
      c1     = cc
      h      = integ(v, h0)
      v      = integ(a, v0)
      a      = (1.0/m) * (thrust - m * g)
      m      = integ(mdot, m0)
      mdot   = -c1*abs(thrust)
      g      = c2/(h + r) **2
      schedule shutlg .xp. 9934.0 - h
      schedule shutsm .xp. 15.0 - h
    END $ "of DERIVATIVE"
    DISCRETE shutlg
      ff = f2
      cc = c12
    END $ "of DISCRETE shutlg"
    DISCRETE shutsm
      ff = 0.0
      cc = 0.0
    END $ "of DISCRETE shutsm"
    termt (t.ge.tmx .or. h.le.0.0  .or. v.gt.0.0)
  END $ "of DYNAMIC"
END $ "of PROGRAM"
```

In ACSL, state–events are fired whenever the specified state condition crosses zero in positive direction (using the ".xp." operator), or whenever the specified state condition crosses zero in arbitrary direction (using the ".xz." operator).

2.9 Model Validation

Validating a given model for a particular purpose is a very difficult issue that we are not ready yet to deal with in general. However, one particular validation technique exists that I would like to discuss at this point, namely the issue of *dimensional consistency checking.*

The concept of dimensional consistency is a trivial one. In any equation, all *terms* must carry the same units. For example, in the equation:

$$a = \frac{1}{m}(thrust - m \cdot g) \tag{2.10}$$

the thrust is a force which is expressed in N or $kg\ m\ sec^{-2}$. We often shall write this as $thrust[N]$ or $[thrust] = N$. The square brackets denote "units of". Consequently, the term $m \cdot g$ must also be expressed in the same units which checks out correctly since $[m] = kg$, and g is an acceleration, and therefore $[g] = m\ sec^{-2}$. Finally, we can check the dimensional consistency across the equal sign: $[a] = m\ sec^{-2}$ which indeed is the same as the dimension of a force divided by the dimension of a mass, since $N\ kg^{-1} \equiv m\ sec^{-2}$. While this principle is surely trivial, I find that one of the most common errors in my students' papers is a lack of dimensional consistency across equations. The principle is obviously so trivial that the students simply don't bother to check it.

While it would be perfectly feasible to design simulation software that checks the dimensional consistency for us, this is not done in the currently used simulation languages. The reason for this is explained below. Let us look at the simplest stable state–space equation:

$$\dot{x} = -x \tag{2.11}$$

Physically, this equation does not make sense since:

$$[\dot{x}] = [\frac{dx}{dt}] = \frac{[x]}{[t]} = [x] \cdot sec^{-1} \tag{2.12}$$

i.e., different units are on the left and on the right of the equal sign. A correct differential equation would be:

$$\dot{x} = a\, x \tag{2.13}$$

where the parameter (or constant) a assumes a value of $a = -1.0\ sec^{-1}$.

If the simulation software would check the dimensional consistency for us, this would force us to code the above differential equation either using a *constant* or a *parameter* properly declared with its units, rather than being able to plug in a number directly. Eventually, it would be useful to have a modeling tool that can check dimensional consistency upon request. In any event, I strongly urge any potential modeler to perform this consistency check regularly.

2.10 Summary

We have introduced the basic syntax of a few current CSSL's, and we have discussed the major problems in formulating continuous system models, such as the algebraic loop problem. By now, the student should be able to code simple problems in any current CSSL. Please, notice that it is *not* the aim of this text to provide the student with the details of any particular language syntax. For that purpose, the textbook should be accompanied by language reference manuals for the chosen simulation languages. We feel that a student should be perfectly capable of studying such a language reference manual on her or his own, and that it is much more important to provide insight into the basic principles of modeling and simulation mechanisms.

References

[2.1] Donald C. Augustin, Mark S. Fineberg, Bruce B. Johnson, Robert N. Linebarger, F. John Sansom, and Jon C. Strauss (1967), "The SCi Continuous System Simulation Language (CSSL)", *Simulation*, **9**, pp. 281–303.

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

[2.3] François E. Cellier (1986), "Combined Continuous/Discrete Simulation — Applications, Techniques and Tools", *Proceedings 1986 Winter Simulation Conference*, Washington, D.C., pp. 24–33.

[2.4] 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.

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

[2.6] Mathworks, Inc. (1987), *Pro–MATLAB with System Identification Toolbox and Control System Toolbox — User Manual*, 21 Eliot St., South Natick, MA 01760.

[2.7] 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.

[2.8] 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.

[2.9] 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.

Bibliography

[B2.1] YaoHan Chu (1969), *Digital Simulation of Continuous Systems*, McGraw–Hill, New York.

[**B2.2**] Charles M. Close, and Dean K. Frederick (1978), *Modeling and Analysis of Dynamic Systems*, Houghton Mifflin Company, Boston, MA.

[**B2.3**] Wolfgang K. Giloi (1975), *Principles of Continuous System Simulation*, Teubner Verlag, Stuttgart, FRG.

[**B2.4**] Granino A. Korn, and John V. Wait (1978), *Digital Continuous-System Simulation*, Prentice-Hall, Englewood Cliffs, N.J.

Homework Problems

[H2.1] Limit Cycle

The following two equations describe a so-called limit cycle:

$$\dot{x} = +y + \frac{k \cdot x \cdot (1 - x^2 - y^2)}{\sqrt{x^2 + y^2}} \qquad (H2.1a)$$

$$\dot{y} = -x + \frac{k \cdot y \cdot (1 - x^2 - y^2)}{\sqrt{x^2 + y^2}} \qquad (H2.1b)$$

The initial conditions are given as:

$$x(t = 0) = x0, \quad y(t = 0) = y0 \qquad (H2.1c)$$

Simulate this system over a period of 10 *sec* with the following four sets of initial conditions:

$$x0 = 0.4, \quad y0 = 1.0 \qquad (H2.1d)$$

$$x0 = 0.0, \quad y0 = 0.5 \qquad (H2.1e)$$

$$x0 = 0.0, \quad y0 = 0.001 \qquad (H2.1f)$$

$$x0 = 1.0, \quad y0 = 1.5 \qquad (H2.1g)$$

The constant k takes a value of $k = 2.0$. Plot all four trajectories in the $[x, y]$-plane on the same graph. An example of what such a graph may look like is given in Fig.8.18 (for a different limit cycle though). For this purpose, I recommend to use ACSL with either Pro-MATLAB or CTRL-C.

[H2.2] Cannon Ball

A famous cannon ball used by the United States forces during the Independence War can be described by the following set of differential equations:

$$\ddot{x} = -R \cdot v \cdot \dot{x} \qquad (H2.2a)$$

$$\ddot{y} = -R \cdot v \cdot \dot{y} - g \qquad (H2.2b)$$

where x denotes the horizontal position, y denotes the vertical position, and v denotes the absolute velocity of the cannon ball. The following constants are needed in the model: $v_0 = 900\ ft/sec$ is the initial velocity, $R = 7.5 \times 10^{-5}\ ft^{-1}$ is the air friction constant, and $g = 32.2\ ft/sec^2$ is the gravity constant.

The cannon was conquered by the Redcoats when they temporarily took Philadelphia in the late fall of 1777. This, of course, created a major security breach. As a consequence, the above model was officially declassified by Congress in 1910.

Simulate the system for various shooting angles (10^o, 15^o, ... 65^o). Each simulation is to be carried out for the duration needed until the cannon ball hits the ground again. Plot all trajectories as functions of time on one sheet of paper.

[H2.3] Hysteresis

Use two differential equations to generate the sine wave function

$$x(t) = 2 * \sin(t) \qquad (H2.3a)$$

Use two state–events to code a vertical hysteresis as shown in Fig.H2.3a. Use ACSL's *LOGD* subroutine to record the corners of the discontinuity immediately before and after the event. Choose $a = 1.0$, $b = -1.0$, $p = 1.0$, and $n = -1.0$. Simulate the system during 10 *sec*, and plot $y(x)$. The resulting curve should look similar to Fig.H2.3.

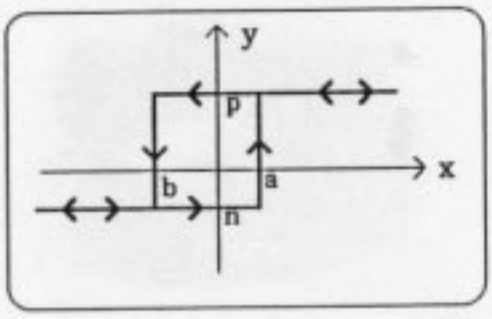

Figure H2.3. Vertical hysteresis function.

[H2.4] Difference Equations

Recode the lunar landing maneuver (the version with scheduled time-events) into a set of difference equations using the Forward Euler integration algorithm presented in this chapter with a fixed step size of 0.1 *sec*.

Code the resulting set of non-linear difference equations in ACSL, and simulate. Compare the results with those presented in the chapter.

[H2.5] Bouncing Ball

A ball is dropped from an altitude of 2 *m*. No initial velocity is applied. The ball falls simply due to the gravitational force. When the ball hits the floor, it bounces. It is assumed that the coefficient of restitution is $k = 0.7$, i.e., the velocity immediately after the impact is exactly 70% of what it was just before the impact, but with reversed sign.

Simulate the system over 10 bounces, and plot the altitude of the ball versus time. Recompute the communication interval during each bouncing event such that 10 communication intervals lie between any two subsequent bounces (this problem is so simple that the time of the next bounce can be computed analytically once the new initial velocity is known). Use ACSL's *LOGD* function to record the times of impact.

[H2.6] Model Validation

Determine a set of consistent measurement units for all constants of the lunar lander model. Verify the dimensional consistency across all of the equations contained in the lunar lander model.