CHAPTER 1

DYNAMIC-SYSTEM MODELS AND SIMULATION

COPYRIGHTED MATERIAL

SIMULATION IS EXPERIMENTATION WITH MODELS

1-1. Simulation and Computer Programs

Simulation is experimentation with models. For system design, research, and education, simulations must not only construct and modify many different models but also store and access a large volume of results. That is practical only with models programmed on computers [1, 2].

In this book we model changes of system variables with time; we represent physical time by the simulation time variable **t**. Our models then attempt to predict different time histories **y1 = y1(t)**, **y2 = y2(t)**, . . . of system variables such as velocity, voltage, and biomass. ***Static models*** simply relate values of system variables **x(t), y(t), . . .** at the same time **t**; a gas pressure **P(t)**, for instance, might be a function **P = aT** of the slowly changing temperature **T(t)**.

Dynamic-system models predict values of model-system ***state variables*** **x1(t)**, **x2(t)**, . . . by relating them to past states **[x1(t), x2(t), . . .]** (Sec. 1-2). Computer simulation of such systems was applied first in the aerospace industry. Simulation is now indispensable not only in all engineering disciplines, but also in biology, medicine, and agroecology. At the same time, discrete-event simulation gained importance for business and military planning.

Simulation is most effective when it is combined with mathematical analyses. But simulation results often provide insight and suggest useful decisions where exact analysis is difficult or impossible. This was true for many early control-system optimizations. As another example, Monte Carlo simulations simply measure statistics over repeated experiments to solve problems too complicated for explicit probability-theory analysis. All simulation results must eventually be validated by real experiments, just like analytical results.

Computer simulations can be speeded up or slowed down for the experimenter's convenience. One can simulate a flight to Mars or to Alpha Centauri in one second. Periodic clock interrupts synchronizing suitably scaled simulations with real time permit "hardware in the loop": One can "fly" a real autopilot—or a human pilot—on a tilt table controlled by computer flight simulation. In this book we are interested

Advanced Dynamic-System Simulation: Model Replication and Monte Carlo Studies, Second Edition. Granino A. Korn.
© 2013 John Wiley & Sons, Inc. Published 2013 by John Wiley & Sons, Inc.

in very fast simulation because we need to study many different model changes very quickly. Specifically, we would like to

- ***enter and edit programs*** in convenient editor windows.
- use ***typed or graphical-interface commands*** to start, stop, and pause simulations, to select displays, and to make parameter changes. Displays of simulation results ought to ***appear immediately*** to provide an intuitive "feel" for the effects of model changes (***interactive modeling***).
- ***program*** systematic parameter-optimization studies and produce cross-plots and statistics.

1-2. Dynamic-System Models

(a) Difference-Equation Models[1]

The simplest way to relate present values **x(t)** and past values **x(t − Δt)** of a ***state variable*** **x = x(t)** is a difference equation such as the simple recurrence

$$\mathbf{x(t) = F[x(t), x(t - \Delta t)]}$$

More general difference-equation models relate several state variables and their past values. In Chapter 2 we discuss such models in detail.

(b) Differential-Equation Models

Much of classical physics and engineering is based on ***differential-equation models*** that relate delayed interactions of continuous ***differential-equation state variables*** **x1(t), x2(t), . . .** with first-order ordinary differential equations (***state equations***)[2]

$$\mathbf{(d/dt)\ xi = fi(t; x1, x2, \ldots; y1, y2, \ldots; a1, a2, \ldots)} \qquad \mathbf{(i = 1, 2, \ldots)} \qquad (1\text{-}1a)$$

Here **t** again represents the time, and the quantities

$$\mathbf{yj = gj(t; x1, x2, \ldots; y1, y2, \ldots; b1, b2, \ldots)} \qquad \mathbf{(j = 1, 2, \ldots)} \qquad (1\text{-}1b)$$

are ***defined variables***. **a1, a2, . . .** and **b1, b2, . . .** are constant ***model parameters***.

A computer-implemented ***simulation run*** exercises such a model by solving the state-equation system (1-1) to produce time histories of the system variables **xi = xi(t)** and **yj = yj(t)** for **t = t0** to **t = t0 + TMAX**. An ***integration routine*** increments the model time **t** and integrates the derivatives (1-1a) to produce successive values of **xi(t)** (Sec. 1-7), starting with given initial values **xi = xi(t0)**.

[1] We refer to recursive relations in general as difference equations, whereas some authors reserve this term for relations formulated in terms of explicit finite differences [11].

[2] We reduce higher-order differential equations to first-order systems by introducing derivatives as extra state variables. Thus, $d^2x/dt^2 = -kx$ becomes

$$\mathbf{dx/dt = xdot} \qquad \mathbf{dxdot/dt = -kx}$$

(see also Sec. 1-10).

Each state variable **xi** is a model output. There are three types of defined variables **yj**:

1. model inputs (specified functions of the time **t**),
2. model outputs, and
3. intermediate results needed to compute the derivatives **fi**.

The defined-variable assignments (1-1b) must be sorted into a procedure that derives updated values for all **yj** from current values of the state variables **xi**, already computed **yj** values, and/or **t** without "algebraic loops" (Sec. 1-9).

Some dynamic systems (e.g., systems involving linkages in automotive engineering and robotics) are modeled with differential equations that cannot be solved explicitly for state-variable derivatives as in Eq. (1-1a). Simulation then requires solution of algebraic equations at each integration step. Such ***differential-algebraic-equation systems*** are not treated in this book. References 6 to 11 describe suitable mathematical methods and special software.

(c) Discussion

Much of classical physics (Newtonian dynamics, electrical-circuit theory, chemical reactions) uses differential equations. As a result, most legacy simulation programs are basically differential-equation solvers and relegate difference equations to accessory "procedural" program segments. Modern engineering systems, though, often involve digital controllers and thus sampled-data operations that implement difference equations. In this book we introduce a program package specifically designed to handle such problems. We start with differential-equation problems in Chapter 1 and go on to difference equations and mixed continuous/sampled-data models in Chapter 2.

1-3. Experiment Protocols Define Simulation Studies

Effective computer simulation is not simply a matter of programming model equations. It must also be truly convenient to modify models and to try many different experiments (see also Sec. 1-5). In addition to program segments that list model equations such as those in Sec. 1-2, every simulation needs an ***experiment-protocol program*** that sets and changes initial conditions and parameters, calls differential-equation-solving simulation runs, and displays or lists solutions.

A simple experiment protocol implements a sequence of successive commands: say

```
a = 20.0  |  b = −3.35   (set parameter values)
x = 12.0  (set the initial value of x)
drun      (make a differential-equation-solving simulation run)
reset     (reset initial values)
a = 20.1  (change model parameters)
b = b − 2.2
drun      (try another run)
.........................
```

Each **drun** command calls a new simulation run. The command **reset** resets initial conditions for new runs.

A ***command intepreter*** executes typed commands immediately. Users can inspect the solution output after each simulation run and then enter new commands for another run. Command-mode operation permits interactive programming and program debugging [2].

Graphical-user-interface (GUI) simulation programs replace typed commands with windows for entering model parameters and menus and/or buttons for executing such commands as **run** and **reset** using mouse clicks. This is convenient for special-purpose simulation programs with simple experiment protocols. Typed and programmed commands entered in a console window (command window) permit a much wider choice of operations.

A programmed ***simulation study*** combines experiment-protocol commands into a stored program called an ***experiment-protocol script***. Such a program can branch and loop to call repeated simulation runs (e.g., for parameter optimization or statistical studies). Proper experiment-protocol scripts require a full-fledged computer language with functions, procedures, program loops, conditional execution, and file operations.

Simulation studies can involve many model and parameter changes, so program execution must be prompt and fast. We can ***interpret*** experiment-protocol scripts. But "dynamic" program segments that implement simulation runs update system variables hundreds or thousands of times. Such time-critical operations must be ***compiled***.[3]

1-4. Simulation Software

Equation-oriented simulation programs such as ACSL™ accept model equations in a more or less human-readable notation, sort defined-variable assignments as needed, and feed the sorted equations to a Fortran or C compiler [1]. Berkeley Madonna and Desire (see below) have runtime equation-language compilers and execute immediately. ***Block-diagram interpreters*** (e.g., Simulink™ and the free open-source Scicoslab program) let users compose block-diagram models on the display screen. Such programs execute interpreted simulation runs immediately but relatively slowly. To improve computing speed, most block-diagram interpreters admit precompiled equation-language blocks for complicated expressions, and production runs are sometimes translated into C for faster execution. Alternatively, ACSL, Easy5™, and Berkeley Madonna have ***block-diagram preprocessors*** for compiled simulation programs. Differential-algebraic (DAE) models need substantially more complicated software, preferably using the Modelica Language [3–6]. Dynasim™ and Maplesim™ are examples.

[3] ***Interpreter*** programs translate individual commands one-by-one into the computer's machine language. ***Compilers*** speed program execution by translating complete program segments.

TABLE 1-1. Desire Under Windows

Easy Installation
Simply copy or unzip the distribution folder **mydesire** on a hard disk or flash stick to produce ***a complete, ready-to-run simulation system*** with an editor, help files, graphics, and many user-program examples. Deleting the installation folder un-installs the package without leaving any trace. Unlike most Windows programs Desire never involves the Windows registry.

Run a User Program
- Double-click the **Wdesire.bat** icon (or a shortcut icon) to open a ***Command Window*** and an empty ***Editor Window*** (Fig. 1-la).
- Drag a user-program icon into the Editor window to load the program for editing.
- Clicking the editor's **OK** button transfers the edited program to Desire, and a typed **erun** (or more simply **zz**) command starts execution.

The ***Graph Window*** displays solution graphs. The Command Window shows error messages and output listings.

Additional Editor Windows can be added by typed commands. Multiple Editor Windows let you run and compare different programs, or different versions of the same program (Fig. 1-la).

1-5. Fast Simulation Program for Interactive Modeling

The simulation programs in this book employ the open-source Desire program[4] on the book CD.[5] Command scripts and model descriptions use a natural mathematical notation similar to Basic: for example,

```
y = a * cos(x) + b          d/dt x = −x + 4 * y
```

so that the system is easy to learn. ***You can run all our program examples and make simple parameter changes without learning language details*** (Table 1-1). The Reference Manual on the CD describes Desire operations in detail, and Ref. 2 is an elementary textbook. Sections 1-10 to 1-12 list simple example programs.

Desire runs under Windows™, Linux, and Unix and solves up to 40,000 differential equations in scalar and vector form. Difference equations are handled equally well. Double-precision floating-point arithmetic is used throughout.

The dual-monitor displays in Fig. 1-1 show Desire running under Windows, Linux, and Unix. Programs are entered and edited in ***editor windows*** and controlled by commands typed into a ***command window***. Solution graphs display in a ***graph window***. The graphs in Fig. 1-1 are black-on-white for publication, but ordinarily, different curves are displayed in bright colors.

Each Desire program begins with an interpreted experiment-protocol script that defines the ***experiment***. Subsequent ***DYNAMIC program segments*** define ***models*** that generate time-history output. When the experiment-protocol script encounters a **drun** statement, a built-in runtime compiler automatically compiles a DYNAMIC

[4] *Desire* stands for "direct executing simulation in real time."

[5] Updated versions of the program package can be downloaded without charge from www.sites.google.com/site/gatmkorn.

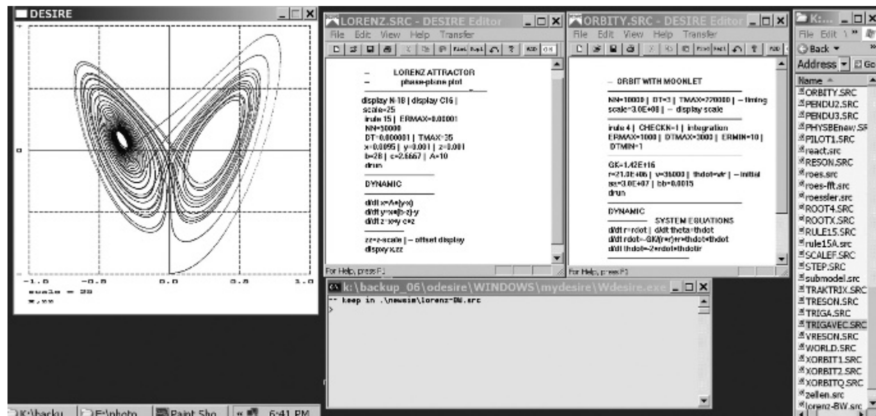


FIGURE 1-1a. Desire running under Windows. The dual-monitor display shows the command window, a file-manager (explorer) window, two editor windows, and the graphics window. The red **OK** button on each Desire editor window transfers the edited program to Desire. Multiple editor windows let you run and compare two or more programs, or modified versions of the same program.

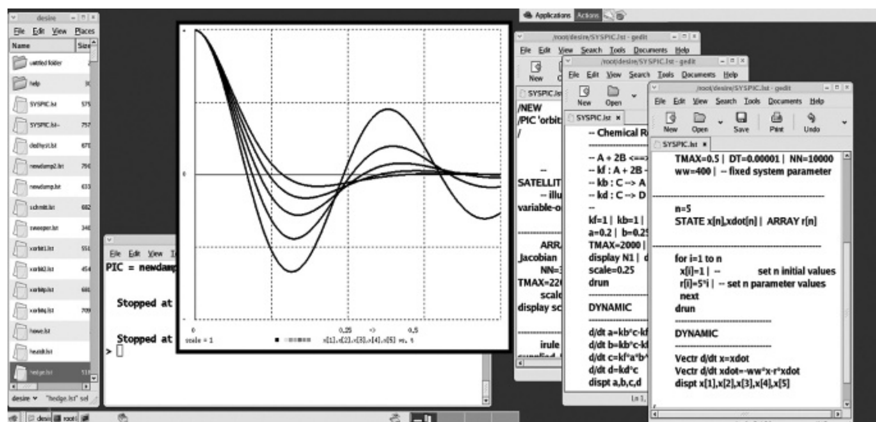


FIGURE 1-1b. Desire running under Linux, showing the command, file-manager, and graphics windows and three editor windows. The Linux Editor's **Save** button transfers the edited program to Desire, just like the **OK** button in Fig. 1-1a.

program segment.[6] A simulation run solving the state equations then executes at once and displays solution time histories.

Fast runtime compilation (under 40 ms) permits truly ***interactive modeling*** since results of screen-edited program changes appear immediately. Multiple editor windows let users enter, edit, and simulate different models to compare results. Runtime displays show solution time histories and error messages during rather than

[6] Any subsequent **drun** call would omit the compilation and simply execute another simulation run.

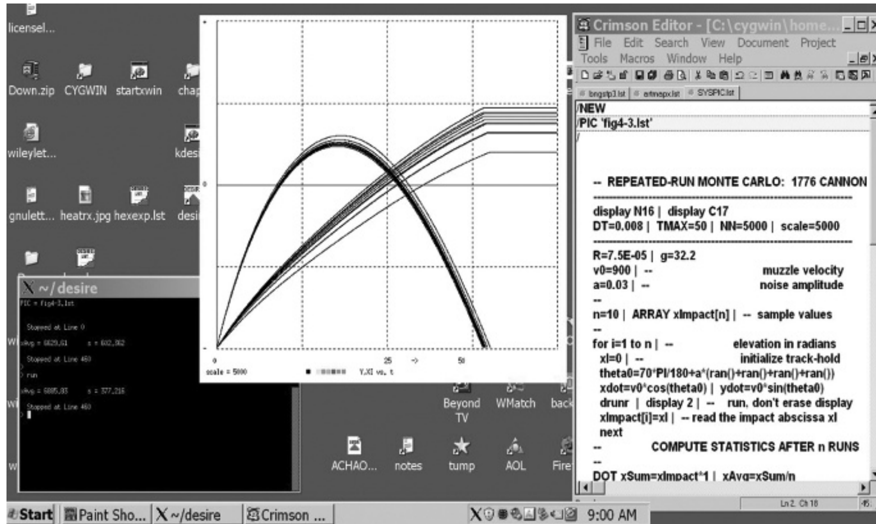


FIGURE 1-1c. Cygwin (Unix under Windows) display with a Unix console window serving as the Desire command window. The single editor window uses the open-source Crimson Editor.

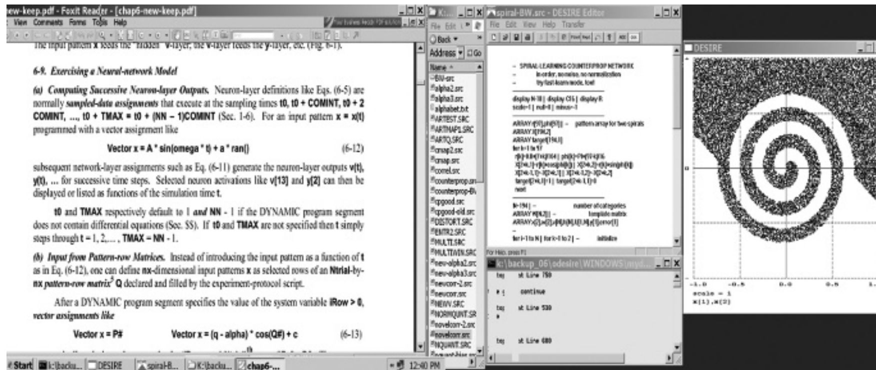


FIGURE 1-1d. Dual-screen display that lets you read textbook pages on the left and run live Desire simulation examples on the right.

after each simulation run. This lets users save time by aborting undesirable runs before they complete.

Experiment protocols can call multiple DYNAMIC segments with different models, different versions of the same model, and/or different input/output operations.

Table 1-1 shows how to run Desire and our program examples under Windows. Under Linux, Desire also installs simply by unzipping a distribution folder. Desire then uses a Linux editor rather than its own editor. The Reference Manual describes the editor installation and its association with user-program text files. Once this is

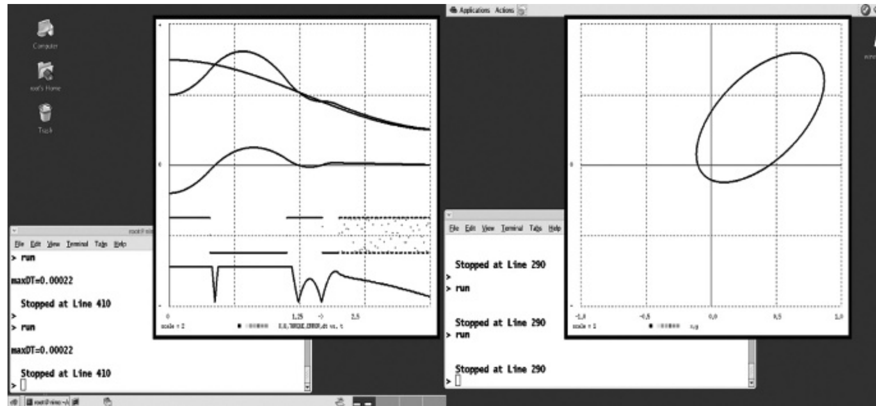


FIGURE 1-1e. Here Linux displays two simultaneous simulations controlled by separate command windows. Multiple file-manager and editor windows could be added.

done, simply clicking a program-file icon displays the program in an editor window, ready to run as under Windows (Fig. 1-1b).

ANATOMY OF A SIMULATION RUN

1-6. Dynamic-System Time Histories Are Sampled Periodically

When **drun** calls a simulation run, the program initializes input/output operations specified in the DYNAMIC program segment. The simulation time **t** and the differential-equation state variables start with initial values assigned by the experiment protocol.[7] A first pass through the DYNAMIC-segment code (1-1) next produces the resulting initial values of the defined variables (1-1b). Unless stopped, the simulation then runs from **t = t0** to **t = t0 + TMAX**. One can ***pause*** a simulation run with a mouse click (Windows) or by typing **ctrl c** and **space** (Linux), and restart or extend a run with **drun**.

Desire normally samples DYNAMIC-segment variables for output or sampled-data operations at **NN** uniformly spaced ***sampling times*** (communication times)

$$\mathbf{t = t0, t0 + COMINT, t0 + 2\,COMINT, \ldots, t0 + (NN - 1)COMINT = t0 + TMAX}$$

$$\mathbf{with \quad COMINT = TMAX/(NN - 1)} \qquad (1\text{-}2)$$

The experiment-protocol script sets appropriate values of **t0**, **TMAX**, and **NN** or uses default values listed in the Reference Manual.

If the DYNAMIC program segment contains differential equations (**d/dt** or **Vectr d/dt** statements), **t0** defaults to **t0 = 0** if no other value is specified. Starting at **t = t0**, the integration routine then increments **t** by successive constant or variable

[7] Unspecified initial values of differential-equation state variables conveniently default to 0.

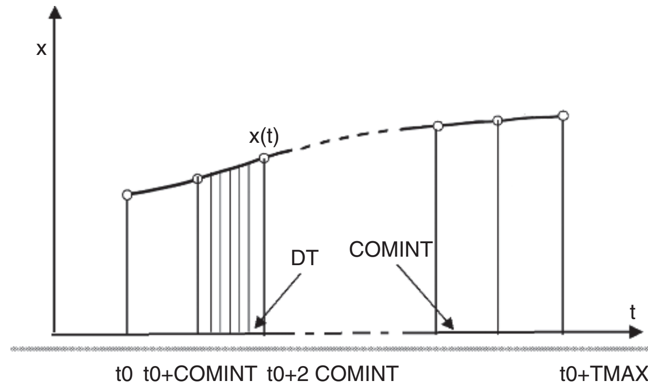


FIGURE 1-2a. Time history of a simulation variable, showing sampling times **t = t0, t0 + COMINT, t0 + 2COMINT, . . . , t0 + TMAX** and some integration steps. In the figure all integration steps end on a sampling point. That is always true for variable-step integration rules, but fixed integration steps **DT** may overshoot the sampling points by a small fraction of **DT**, as shown in Fig. 1-2b.

DT steps until **t** reaches the next data-sampling communication point (Fig. 1-2a). Within integration steps; numerical integration approximates continuous updating of the "continous" model variables **t**, **xi**, and **yj**. Each integration step usually requires more than one ***derivative call*** executing the model equations (1-1) (Sec. 1-7 and Refs. 3 to 11).

In DYNAMIC program segments ***without*** differential equations, **t0** defaults to **t0 = 1** unless the experiment-protocol script specifies a different value. All operations in such a DYNAMIC segment are then ***sampled-data assignments*** and execute

Variable-step integration
NN = 6 | TMAX = 10 | initial DT = 0.01

t,	x,X,y		
0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00
2.00000e+00	3.89418e-01	3.89418e-01	0.00000e+00
4.00000e+00	7.17356e-01	3.89418e-01	3.89418e-01
6.00000e+00	9.32039e-01	9.32039e-01	3.89418e-01
8.00000e+00	9.99574e-01	9.32039e-01	9.32039e-01
1.00000e+01	9.09298e-01	9.09298e-01	9.32039e-01

Fixed-step integration
NN = 6 | TMAX = 11 | initial DT = 0.01

t,	x,X,y		
0.00000e+00	0.00000e+00	0.00000e+00	0.00000e+00
2.00000e+00	3.89419e-01	3.89419e-01	0.00000e+00
4.01000e+00	7.18748e-01	3.89419e-01	3.89419e-01
6.01000e+00	9.32762e-01	9.32762e-01	3.89419e-01
8.01000e+00	9.99513e-01	9.32762e-01	9.32762e-01
1.00100e+01	9.08463e-01	9.08463e-01	9.32762e-01

FIGURE 1-2b. Desire output listings for variable-step integration and for fixed-step integration. Parameters were deliberately chosen to exaggerate the fixed-**DT** effect.

at successive communication times (1-2). Assignments preceded by a **SAMPLE m** statement, where **m** is an integer **>** 1, execute only at **t = t0 + COMINT** and then at every **m**th communication point. This permits multirate sampling.

DYNAMIC-segment ***input/output*** (e.g., output to displays and listings) occurs at the **NN** communication points (1-2) unless the system variable **MM**, which defaults to 1, is set to an integer **>** 1. In that case, input/output occurs at **t = t0 + COMINT** and then at every **MM**th sampling point, and finally at **t = t0 + TMAX**. **NN** can thus be set to a larger value than the desired number of input/output points. This can provide fast sampling for pseudorandom noise (Sec.4-14) and/or for sampling switch and limiter functions (Secs. 2-10 and 2-11).

Some defined-variable assignments (1-1b) do not affect state variables but only scale or modify model output. Such operations are not needed at every derivative call but only at sampling points. Simulations run faster if one programs such assignments as sampled-data operations following an **OUT** statement.

Finally, Desire is designed to solve problems that combine differential equations with difference equations. Differential-equation-solving DYNAMIC segments can include difference-equation code that must not execute in the middle of integration steps. In particular, ***sampled-data assignments*** modeling digital controllers and noise generators execute only at periodic sampling points and must be collected in sections following an **OUT** and/or **SAMPLE m** statement at the end of the DYNAMIC program segment. Nonperiodic difference-equation code (recursive assignments) must similarly follow a **step** statement. These topics are discussed in Chapter 2.

1-7. Numerical Integration (see also Table A-1)

(a) Euler Integration

The simplest procedure that approximates continuous updating of a state variable **x** in successive integration steps is the ***explicit Euler integration rule***

xi(t + DT) = xi(t) + fi[t; x1(t), x2(t), . . . ; y1(t), y2(t), . . .] DT (i = 1, 2, . . . , n) (1-3)

where **fi** is the value of **dx/dt** calculated by the derivative call executing Eq. (1-1) at the time **t**.

The integration routine loops until **t** reaches the next communication point (1-2), where the solution is sampled for input/output and sampled-data operations. The simulation run terminates after accessing the last sample at **t = t0 + TMAX**, unless the run is stopped either by the user or by a programmed termination (**term**) statement.

(b) Improved Integration Rules [6–11]

The Euler integration rule (1-3) simply increments each state variable by an amount proportional to its last-computed derivative. That is an acceptable approximation to true integration only for very small integration steps **DT**. Improved updating requires multiple derivative calls per integration step **DT**. This can reduce the total number of derivative calls (the main computing load of a simulation) required for a specified accuracy. In particular,

- ***multistep rules*** extrapolate updated **xi** values as polynomials based on values of the **x1**, **x2**, . . . and **f1**, **f2**, . . . at several past times **t − DT, t − 2 DT, . . .**.
- ***Runge–Kutta rules*** precompute two or more approximate derivative values in the interval **(t, t + DT)** by Euler-type steps and use their weighted average for updating.

Coefficients in such integration formulas are chosen so that polynomials of degree **N** integrate exactly (**N**th-order integration formula).

Explicit integration rules such as Eq. (1-3) express future values **xi(t + DT)** in terms of already computed past state-variable values. ***Implicit*** rules such as the ***implicit Euler rule***

$$\mathbf{xi(t+DT) = xi(t) + fi[t+DT; x1(t+DT), x2(t+DT), ...; \; y1(t+DT), y2(t+DT), ...] \, DT} \quad \mathbf{(i = 1, 2, ..., n)} \qquad (1\text{-}4)$$

require a program that solves the predictor equations (1-4) for the **xi(t + DT)** at each integration step. This clearly involves more computation. But implicit integration rules often produce more stable solutions and may admit larger **DT** values without numerical instability, and thus still save computing time.

Variable-step integration adjusts integration step sizes to maintain accuracy estimates obtained by comparing various tentative updated solution values. This can save many steps. Figures 1-5, 8-7, and 8-8 show examples.

Numerical integration normally assumes integrands **fi** that are continuous and differentiable within each integration step. Step-function inputs are acceptable only at **t = t0** and thereafter at the end of integration steps. This problem is discussed in Sections 2-9 to 2-11 in connection with models involving sampled-data operations and switching functions.

1-8. Sampling Times and Integration Steps

The experiment protocol script selects the simulation-run time **TMAX** and the number of samples **NN** needed for display, listings, and/or sampled-data models. Desire returns an error message if you select an integration-step value **DT** value larger than **COMINT = TMAX/(NN − 1)**; Desire ***never samples data within integration steps***.[8] Sampled-data output to displays or sampled-data assignments is not well defined at such times. Sampled-data ***input*** within integration steps might make the numerical-integration routine invalid (see also Secs. 2-9 to 2-12).

Desire's variable-step integration routines automatically force the last integration step in each communication interval to end precisely on one of the user-selected communication points (1-2). An "illegal sampling rate" message warns you if the initial **DT**-value exceeds **COMINT**. Fixed-step integration routines, though, may have to add a fraction of **DT** to each sampling time (1-2) to make sure that sampling always occurs at the end of an integration step, as shown in Fig. 1-2b. This does not cause errors in displays or listings, for each **x(t)**-value is still associated with its correct

[8]Some other simulation programs admit larger **DT** values and produce output within integration steps by interpolation. The accuracy of the interpolation routine must match that of the integration routine.

t-value. But to produce output listings at exactly specified periodic sampling times (1-2) you must either use variable-step integration or set **DT** to a very small integral fraction of **COMINT**.

1-9. Sorting Defined-Variable Assignments

DYNAMIC-segment operations (1-1) preceding an **OUT** or **SAMPLE m** statement (if any) execute at every call of the differential-equation-solving integration routine. Each derivative or defined-variable assignment uses the time and state-variable values computed by the last derivative call. Derivative and defined-variable values for **t = t0** are derived from the given initial state-variable values and **t0** by an extra initial derivative call.

The state equations (1-1a) are normally programmed following the defined-variable assignments (1-1b). The defined-variable assignments may use **yj**-values already computed in the course of the current step. They must, therefore, execute in the correct procedural order to derive each **yj**-value from the current state-variable values and **t**. An out-of-order assignment might not find all its arguments, or try to use defined-variable values from an earlier derivative call. Legacy differential-equation solvers such as ACSL sort the defined-variable assignments automatically so that they use only **yi**-values already computed by the current derivative call. If that is impossible due to an ***algebraic loop***, the program returns an error message (***sort error***).

Since unlike most legacy differential-equation solvers, Desire accommodates difference equations directly (Chapter 2), we do not sort defined-variable assignments automatically. As-yet-undefined variables are correctly identified with error messages, but users must inspect algebraic loops (if any) and re-sort the assignments if necessary.

Desire does not treat recursive assignments such as **qi = Fi(t; qi)** as algebraic loops but recognizes them automatically as difference equations (Sec. 2-2). In Secs. 2-16 to 2-21 we discuss significant applications of this technique.

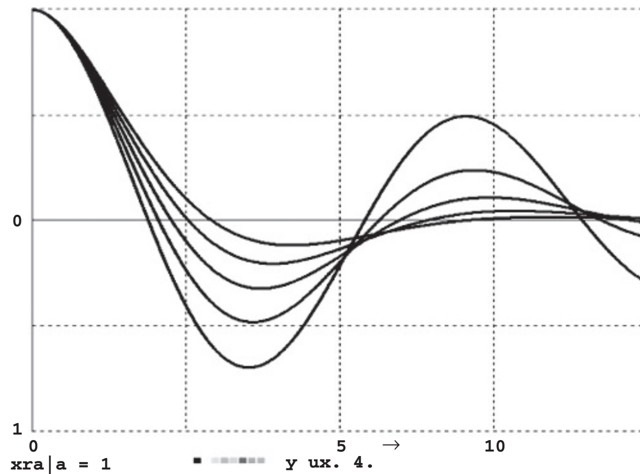
SIMPLE APPLICATION PROGRAMS

1-10. Oscillators and Computer Displays

(a) Linear Oscillator

The complete small program in Fig. 1-3 illustrates the main features of a Desire simulation. The ***DYNAMIC program segment*** following the **DYNAMIC** statement in Fig. 1-3a defines a differential-equation model. We modeled a simple damped harmonic oscillator or mass–spring–dashpot system with the derivative assignments

d/dt x = xdot | d/dt xdot = −ww * x − r * xdot




```
--                          A LINEAR OSCILLATOR
------------------------------------------------
TMAX = 10 | DT = 0.0001 |  NN = 10001
ww=0.8  | --                    parameter value
x = 1  | --                        initial value
------------------------------------------------
for i = 1 to 5  | --       set parameter values
  r = 0.2 * i
  drunr | display 2  | --  don't erase display
  next
------------------------
DYNAMIC
------------------------
d/dt x = xdot |  d/dt xdot = - ww * x - r * xdot
dispt x
```

FIGURE 1-3a. Complete simulation program for a linear oscillator, producing five simulation runs with different values of the damping coefficient **r**.

We can add a ***display specification:***

- **dispt x, xdot** displays the variables **x** and **xdot** versus the simulation time **t**
- **dispxy x, xdot** displays **xdot** versus **x** (***phase-plane plot***)

Model and display are exercised by the ***experiment-protocol script*** preceding the **DYNAMIC** statement. In Fig. 1-3a successive experiment-protocol lines specify

- the runtime **TMAX**, the integration step **DT**, and the number **NN** of display points
- a model parameter **ww**
- the initial value of the state variable **x**

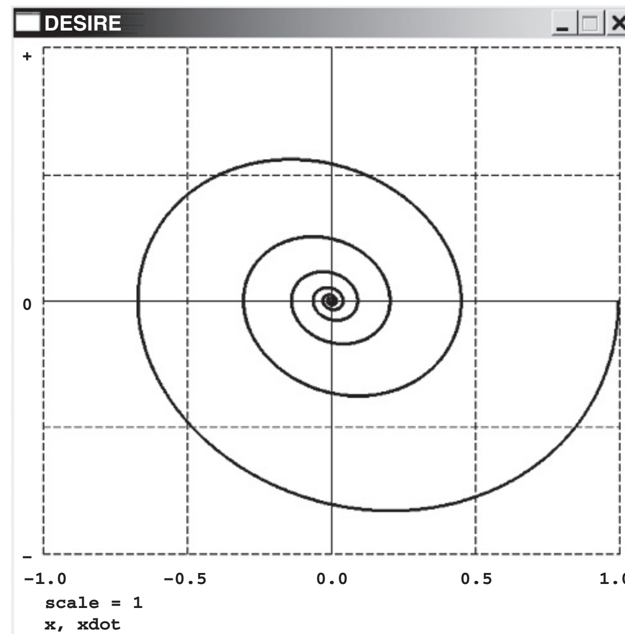


FIGURE 1-3b. Phase-plane plot (**xdot** vs. **x**) for the linear oscillator in Fig. 1-3a.

Initial values of the time **t** and of the state variable **xdot** were not specified and default to 0. The integration routine defaults to a fixed-step second-order Runge–Kutta rule.[9]

A simple experiment-protocol loop next calls five simulation runs with five values of the oscillator damping parameter **r**. The **display 2** statement keeps the display alive through multiple runs. The resulting displays are reproduced at the top of 1-3a. Figure 1-3b shows a phase-plane plot.

(b) Nonlinear Oscillator: Duffing's Differential Equation

The differential equations

d/dt x = xdot | d/dt xdot = −x * x * x − a * xdot

model an oscillator with a nonlinear spring. Figure 1-4a and b show the resulting time histories and phase-plane plots obtained with **a = 0.02**. These results are clearly different from the linear-oscillator response in Fig. 1-3.

If we drive the nonlinear oscillator with a sinusoidal voltage **b cos(t)**, we obtain

d/dt x = xdot | d/dt xdot = −x * x * x − a * xdot + b * cos(t)

Figure 1-4b shows solution displays and program. Note that the experiment-protocol script first calls a simulation run to exhibit the initial transient, then a long simulation

[9]The Desire Reference Manual on the book CD describes in detail the complete program syntax, default values of different simulation parameters, and operating instructions.

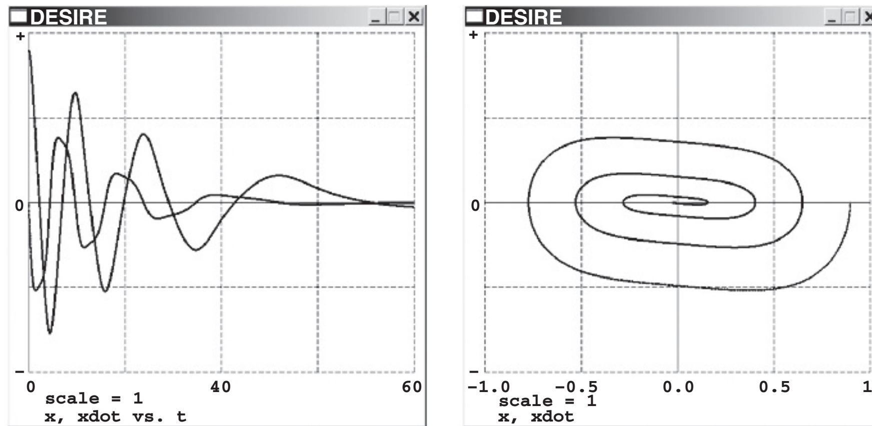


FIGURE 1-4a. Time histories and a phase-plane plot for the nonlinear oscillator modeled with
d/dt x = xdot | d/dt xdot = −x * x * x − a * xdot + b * cos(t)

run with the display turned off to establish steady-state conditions, and finally, a third run to display the steady-state solution.

Reference 2 has more Desire programs for small physics problems.

1-11. Space-Vehicle Orbit Simulation with Variable-Step Integration

The space-vehicle orbit simulation in Fig. 1-5 assumes a fixed Earth that exerts a simple inverse-square-law gravitational force on a satellite. Forces exerted by the Moon are neglected. With Earth at the coordinate origin, the inverse-square-law accelerations in the **x** and **y** directions are

$$(d/dt)\ xdot = -(a/R^2)\ x/R \qquad (d/dt)\ ydot = -(a/R^2)\ y/R$$

The program is scaled so that the gravitational constant **a** equals 1, and we obtain a very simple differential-equation system:[10]

```
rr = (x^2 + y^2)^(-1.5)
d/dt x = xdot   |   d/dt y = ydot
d/dt xdot = -x * rr   |   d/dt ydot = -y * rr
```

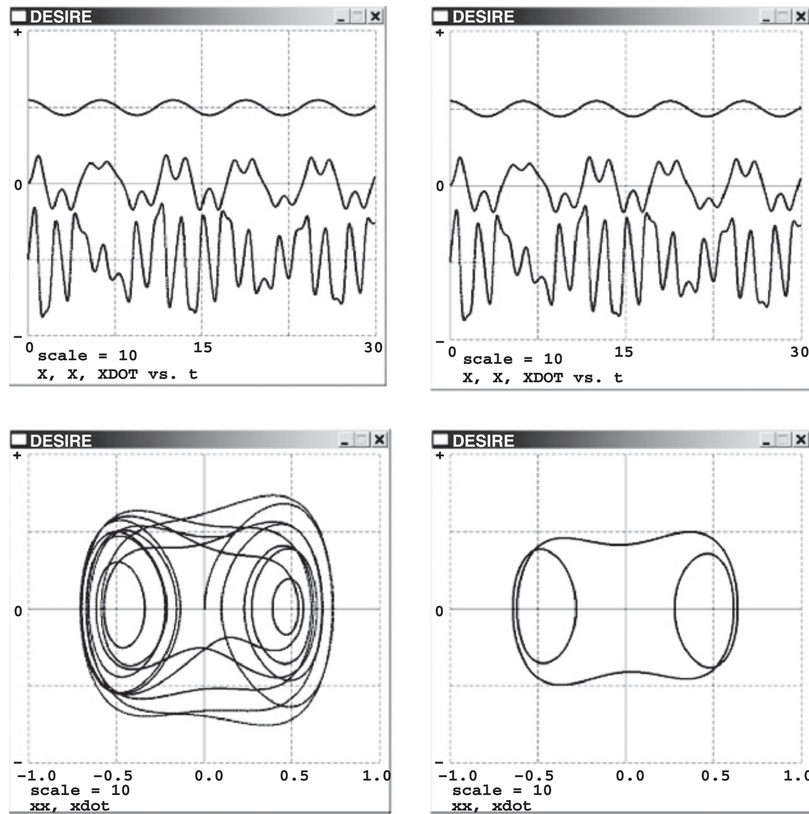
The orbit in Fig. 1-5 involves dramatic velocity changes, and the small integration steps required during the high-velocity portion of the trajectory would slow the

[10]This Cartesian-coordinate formulation is simpler than the polar-coordinate differential-equation system [2]

```
x = r * cos(theta)     |   y = r * sin(theta)
d/dt r = rdot          |   d/dt rdot = −GK/(r^2) + r * thdot^2
d/dt theta = thdot     |   d/dt thdot = 2 * rdot * thdot/r
```

used in Refs. 1 and 2.

JWBS097-c01 JWBS097-Korn Printer: Yet to Come January 16, 2013 8:6 6.125in×9.25in




```
--              DUFFING'S DIFFERENTIAL EQUATION
-------------------------------------------------------------------
scale = 10
TMAX = 30  |  DT = 0.0002  |  NN = 10000
a = 0.099  |  b = 15  |  --                          parameters
X = 0.02  |  --                                      initial value
drun
write "type go to continue"  |  STOP
TMAX = 200  |  display 0  |  drun
write " note how solution becomes periodic!"
TMAX = 30  |  display 1  |  drun
-------------------------------------------------------------------
DYNAMIC
-------------------------------------------------------------------
z = cos(t)
d/dt x = xdot  |  d/dt xdot = - a * xdot - x * x * x + b * z
--
Z = 0.5 * (z + scale)  |  X = 0.5 * x  |  XDOT = 0.5 * (xdot - scale)
dispt Z, X, XDOT  | --       (or use  xx = 2 * x  | dispxy xx,xdot)
```

FIGURE 1-4b. Simulation program for Duffing's differential-equation system. The experiment protocol first calls a simulation run demonstrating the initial transient, then a long run without display to obtain a steady state (**TMAX = 200, display 0**, **display 0**), and finally, a third run showing the steady-state solution with the display turned on again (**display 1**). Phase-plane plots are shown as well. A plot of **z = cos(t)** is shown for comparison.

rest of the simulation. For this reason such simulations employ an implicit variable-step/variable-order integration rule (**irule15**). The second display in Fig. 1-5 illustrates the integration-step changes.

1-12. Population-Dynamics Model

Typical population-dynamics models represent population counts by continuous differential-equation state variables. There can be any number of populations, including subpopulations such as age and gender cohorts. Assignments to the state derivatives describe interactions of different populations that may breed, die, contract diseases, and fight or eat one another. Quite similar state-equation systems also describe the reaction rates of "populations" of chemical compounds or radioactive isotope mixtures (Sec. 8-1).

The classical example of a two-population predator–prey interaction is modeled by the ***Volterra–Lotka differential equations***

```
d/dt prey = (a1 − a4 * predator) * prey
d/dt predator = (−a2 + a3 * prey) * predator
```

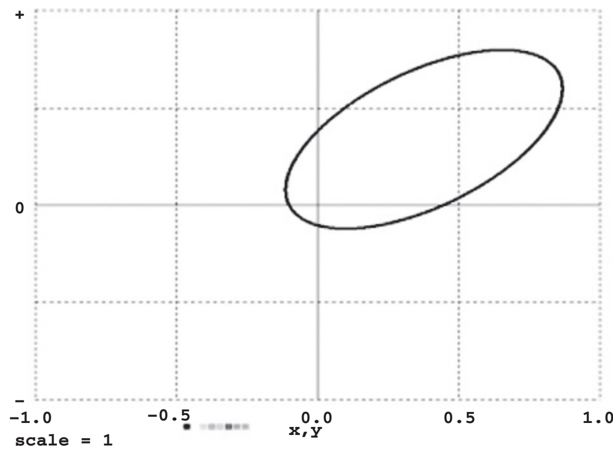
Rates of change of each population are proportional to the population size. **a1** is the difference between the natural birth and death rates of the prey (say, of a local population of rabbits). The prey has an additional death rate **a4 * predator** proportional to the size of the predator population (say, a population of foxes). The predator population has a death rate **a2**, and its birth rate **a3 * prey** is proportional to the prey population, which is its food supply.

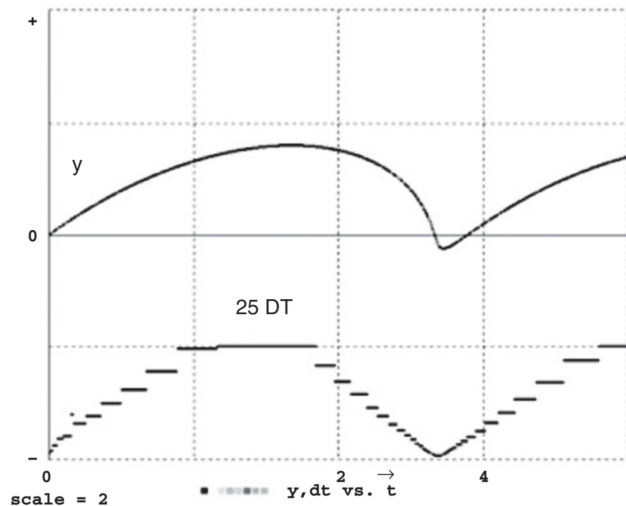
The simulation program in Fig. 1-6 demonstrates how easily such simple population-dynamics models can be modified. We added an extra predator death rate **b * predator** to account for the effect of crowding as the predator population increases and some predators kill one another. For **b = 0** (no crowding) we obtain the classical periodic Volterra–Lotka solution: as the rabbits breed, the foxes have more food; their number increases until they seriously reduce the rabbit population and thus their own food supply. The number of rabbits then increases again, and the process repeats. But crowding (**b > 0**) limits the predator population, and both populations converge to steady-state values.

1-13. Splicing Multiple Simulation Runs: Billiard-Ball Simulation

The DYNAMIC program segment in Fig. 1-7 models a billiard ball as a point **(x, y)** on a table bounded by elastic barriers at **x = a, x = −a, y = b**, and **y = −b**. For **x** and **y** within the barriers, the only acceleration is due to constant friction in the negative velocity direction, so that we program

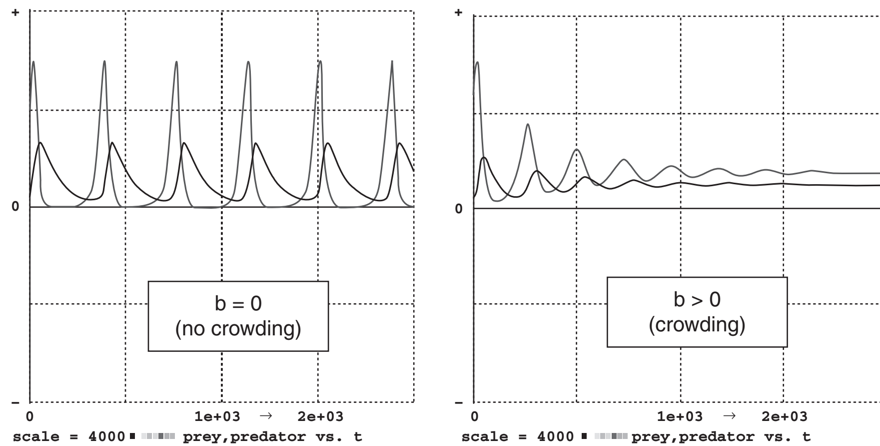
```
d/dt x = xdot    |    d/dt y = ydot
d/dt xdot = −fric * xdot/v    |    d/dt ydot = −fric * ydot/v
```






```
--                        SPACE-VEHICLE-ORBIT SIMULATION
--------------------------------------------------------------------------
irule 15 | ERMAX = 0.0000001 | -- Gear-type integration
xdot = 1.4 | dot = 0.9 | x = 0.45 | y = 0
TMAX = 4 | DT = 0.0001 | NN = 10000
drun
--------------------------------------------------------------------------
DYNAMIC
--------------------------------------------------------------------------
rr = (x^2 + y^2)^(-1.5)
d/dt x = xdot | d/dt y = ydot
d/dt xdot = - x * rr | d/dt ydot = - y * rr
```

FIGURE 1-5. Space-vehicle-orbit simulation program, orbit display, and stripchart time histories of **y** and **DT**, showing the variable integration steps. For simplicity, the problem was scaled so that all coefficients equal unity.




```
--    A PREDATOR-PREY PROBLEM
--                        showing the effect of crowding
------------------------------------------------------------
TMAX = 2000  |  DT = 0.01  |  NN = 5000  |  scale = 4000
a1 = 0.05  |  a2 = 0.01  |  a3 = 2.0E-05  |  a4 = 1.0E-04
b = 0
prey = 2000  |  predator = 200  |  --      initial values
drunr
write " type go to see effect of predator crowding"
STOP
b = 1.0E-05  |  drun
-----------------------------------------------------
DYNAMIC
-----------------------------------------------------
d/dt prey = (a1 - a4 * predator) * prey
d/dt predator = (- a2 + a3 * prey - b * predator) * predator
dispt prey, predator
```

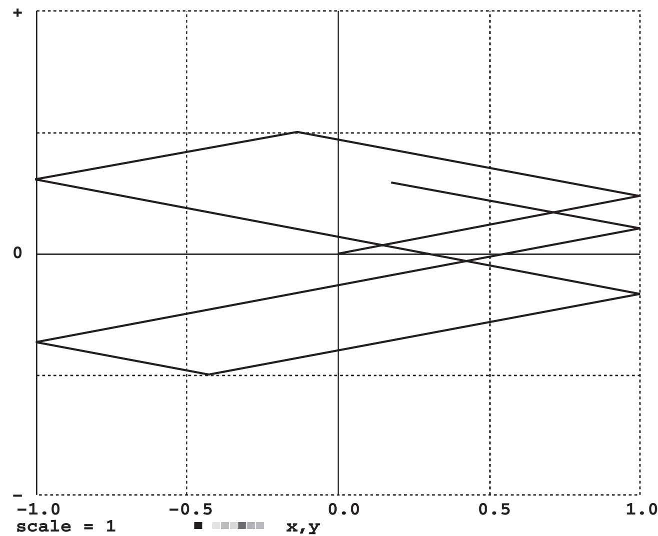
FIGURE 1-6. Population-dynamics simulation. For **b = 0** the program implements the classical Volterra–Lotka differential equations, which produce steady-state periodic fluctuations of the predator and prey populations. Positive values of **b** model an increased predator death rate due to crowding (e.g., by predator cannibalism). Predator and prey populations then converge to constant steady-state values.

where the velocity **v** is obtained with the defined-variable assignment

v = sqrt(xdot^2 + ydot^2)

A differential-equation model of barrier impacts would need to formulate elastic and dissipative forces produced as the ball penetrates each barrier. This is not only complicated but involves very large accelerations and thus small integration steps. We neatly avoid these problems by ***terminating the simulation run*** when a barrier is reached; that is, for **|x| > a** or **|y| > b**:

term abs(x) − a | term abs(y) − b




```
--                                                  BILLIARDS
---------------------------------------------------------------------------
NN = 2000  |  DT = 0.01
TMAX = 20 |  Tstop = 1000
-----
R = 0.9  |  --                          restitution parameter
fric = 0.0005  |  --              acceleration due to friction
a = 1  |   b = 0.5
xdot = 0.15 |  ydot = 0.035
repeat
  drun  |  display 2  |  --           don't erase the display
  if abs(x) >  a then xdot = - R * xdot  |  ydot = R * ydot
    else proceed
  if abs(y) > b then xdot = R * xdot  |  ydot = - R * ydot
    else proceed
  until t > Tstop
---------------------------------------------------------------------------
DYNAMIC
---------------------------------------------------------------------------
v = sqrt(xdot^2 + ydot^2)
d/dt x = xdot    |   d/dt y = ydot
d/dt xdot = - fric * xdot/v   |   d/dt ydot = - fric * ydot/v
term abs(x) - a   |   term abs(y) - b
term t - Tstop
dispxy x,y
```

FIGURE 1-7. Billiard-ball simulation. The experiment-protocol script splices multiple simulation runs terminated by impact on one of four barriers at **x = a, x = −a, y = b, y = −b**.

The experiment-protocol script then starts a new simulation run with the current position coordinates **x** and **y** and "reflected" velocity components **xdot** and **ydot**:

```
if abs(x) > a then xdot = −R * xdot   |   ydot = R * ydot
   else proceed
if abs(y) > b then xdot = R * xdot   |   ydot = −R * ydot
   else proceed
```
(1-5)

where the restitution parameter **R** measures the energy absorbed by the impact. A repeat loop continues this process until **t > Tstop**. The detailed syntax of **if/then/else** and **repeat/until** statements in Desire experiment-protocol scripts is given in the Reference Manual on the book CD. Figure 1-7 shows typical results as friction eventually brings the billiard ball to rest. **display 2** again keeps the program from erasing the display between runs.

Similar run-splicing experiment-protocol scripts are useful in many other applications with radical switching operations, including simulations of electronic switching circuits. Reference 2 exhibits more examples, including the classical bouncing-ball simulation and the EUROSIM peg-and-pendulum and switched-amplifier benchmarks.

INRODUCTION TO CONTROL-SYSTEM SIMULATION

1-14. Electrical Servomechanism with Motor-Field Delay and Saturation

The motor of an electrical servomechanism drives a load so that the output displacement **x** follows a given input **u = u(t)**, typically after an initial transient (Fig. 1-8). The servo controller produces the motor-control voltage **voltage** as a function of the position error **error = x − u** and the rate of change **xdot = dx/dt** measured continuously by a tachometer on the motor shaft.

Figure 1-8 shows a simulation program. Note that the sinusoidal servo input **u = A * cos(w * t)** reduces to a step input for **w = 0**. We model a simple ***linear controller*** with

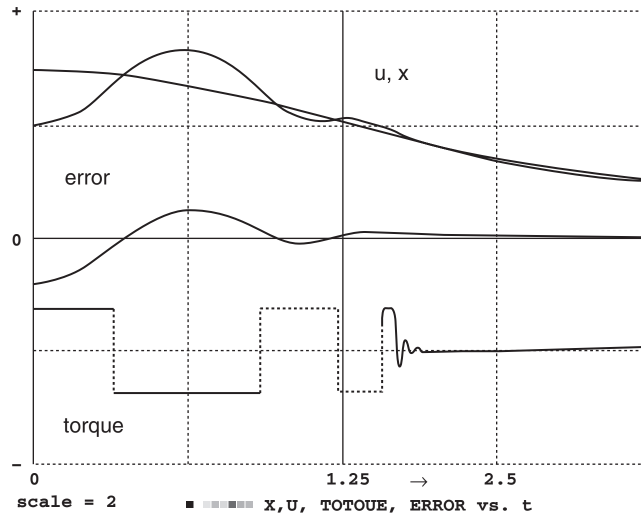
```
voltage = −k * error − r * xdot
```
(1-6)

The controller gain **k** and damping coefficient **r** are positive controller parameters. As is well known, high gain and/or low damping speed the servo response but can cause output overshoot or even oscillations and instability. A ***nonlinear*** controller is discussed in Chapter 8.

The motor voltage (1-6) produces a field current **I** with a field-buildup delay modeled with

```
d/dt I = −B * I + g1 * voltage
```
(1-7)

JWBS097-c01 JWBS097-Korn Printer: Yet to Come January 16, 2013 8:6 6.125in×9.25in




```
--                    SERVOMECHANISM SIMULATION
---------------------------------------------------
scale = 2 | display N1 | display C8 | -- display
TMAX = 2.5 | DT = 0.0001 | NN = 10000
---------------------------------------------------
A = 0.1 | w = 1.2 | --        -- signal parameters
B = 100 | maxtrq = 1.5 | --      motor parameters
g1 = 10000 | g2 = 1 | R = 0.6
k = 40 | r = 2 | --           -- controller parameters
--
drun
---------------------------------
DYNAMIC
---------------------------------
u = A * cos(w * t) | --          input
error = x - u | --           servo error
----------------------------------------------
voltage = - k * error - r * xdot | --   motor voltage
d/dt I = - B * I + g1 * voltage | -- motor field delay
torque = maxtrq * tanh(g2 * I/maxtrq)
d/dt x = xdot | d/dt xdot = torque-R * xdot
--
---------------------------   scaled stripchart display
X = 5 * x + 0.5 * scale | U = 5 * u + 0.5 * scale
ERROR = 4 * error | TORQUE = 0.25 * torque - 0.5 * scale
dispt X,U,TORQUE,ERROR
```

FIGURE 1-8. Complete simulation program and stripchart display for an electrical servo with motor-field delay, field saturation, and sinusoidal input **u = A * cos(w * t)**. You can also set **w = 0** to obtain the servomechanism step response.

The resulting motor torque is limited by motor-field saturation represented by the soft-limiting hyperbolic-tangent function

$$\textbf{torque = maxtrq * tanh(g2 * I/maxtrq)} \tag{1-8}$$

The response of motor, gears, and load to the torque satisfies the differential equations of motion

$$\textbf{(d/dt)x = xdot} \qquad \textbf{(d/dt)xdot = (torque − R * xdot)/M} \tag{1-9}$$

where **M** represents the inertia of motor, gears, and load, and **R > 0** is a motor damping parameter. For convenience, **torque** and **R** are scaled so that **M = 1**.

The simulation program in Fig. 1-8 sets system parameters and models the servomechanism with two defined-variable assignments (1-6) and (1-8) and three state differential equations (1-7) and (1-9). Control-system designers can then exercise the resulting "live mathematical model" to observe servo input, output, error, and motor torque while they adjust controller parameters and motor characteristics. Desirable parameter combinations must, in some sense, produce small servo errors. We can apply different test inputs **u(t)** similar to normal inputs for the intended application (e.g., step inputs, ramps, sinusoids, or noise). Simulations must be repeated with different input amplitudes, since the field saturation makes our model nonlinear.

Such computer-aided experiments provide some intuitive feel for the control problem and may quickly indicate instability or design errors. For objective decision-making, though, we must define and compute numerical ***error measures***. These are typically functionals determined by the entire time history of the servo error **x(t) − u(t)** for a given input **u(t)**. One can, for instance, record the maximum of the ***absolute error*** or of the ***squared error*** as in Sec. 2-16c. More commonly used error measures are integrals over the error time history. We define such measures as extra state variables with zero initial values, for instance

```
d/dt IAE = abs(x − u)            (IAE, integral absolute error)
d/dt ISE = (x − u)^2             (ISE, integral squared error)
d/dt ITAE = t * abs(x − u)
d/dt ISTAE = t^2 * abs(x − u)
```

ISE/TMAX is the ***mean square error*.**

We can now vary the design parameters until selected error measures meet acceptance limits, or until an error measure is as small as possible. We may also want to study our control system's effect on the controlled machine or vehicle (e.g., with a view to minimizing excessive space-vehicle accelerations). Parameter-influence studies are discussed in more detail in Secs. 4-1 to 4-3.

1-15. Control-System Frequency Response

Simulation experiments can explore ***control-system frequency response*** with successive different sinusoidal inputs. Desire experiment-protocol scripts can perform fast Fourier transforms and work with complex numbers for frequency-response and root-locus plots [2]. For a ***linear*** control system we make a differential-equation-solving simulation run and then obtain the frequency response with a fast Fourier transform. We describe such operations in Secs. 8-15 to 8-19 after we acquire more modeling tools in Chapter 3.

1-16. Simulation of a Simple Guided Missile [12–15]

(a) Guided Torpedo

In Fig. 1-9a a missile pursues a target. The problem is scaled so that **TMAX = 1**, and distances are in 1000-f units. **x** and **y** are rectangular Cartesian coordinates of the missile center of gravity; **u** and **v** are velocity components along and perpendicular to the missile longitudinal axis; **phi** is the flight path angle; and **rudder** is the control-surface deflection. The target proceeds on a straight course at constant velocity.

Our particular missile is a guided torpedo. In water, drag and side forces are approximately proportional to the square $\mathbf{u}^2$ of the torpedo velocity **u**. Accelerations along and perpendicular to the torpedo longitudinal axis are then approximated by

(d/dt) u = (thrust − drag)/mass = UT − a2 * u²
(d/dt) v = b1 * u² * sin γ2 + b2 * phidot + b3 * v * rudder

The yaw-rotation equations are

(d/dt) phi = phidot
(d/dt) phidot = c1 * u² * sin γ + c2 * u * phidot + c3 * u² * rudder

c1 and **c2** are hydrodynamic-moment and damping-moment coefficients, and **c3** is the rudder steering-moment coefficient, all divided by the torpedo moment of inertia.

Weathercock stability ensures that the angle of attack **γ2** between the torpedo longitudinal axis and the velocity vector is so small that

sin γ2 ≈ tan γ2 ≈ v/u

and the differential equations of motion for our DYNAMIC program segment become

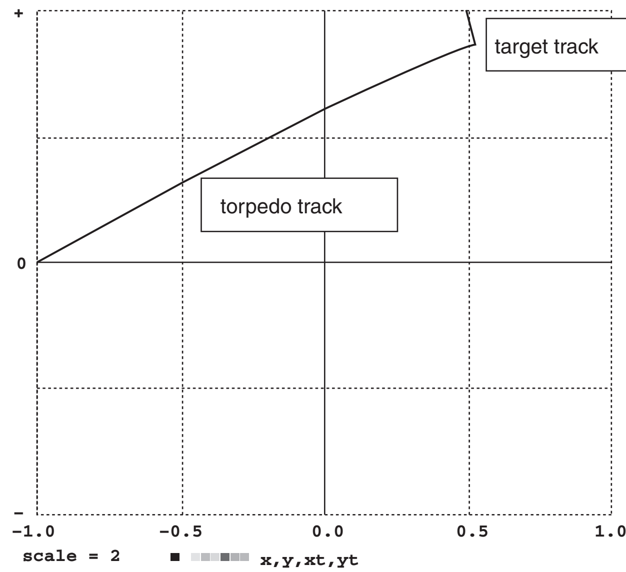
```
(d/dt) u = UT − a2 * u²
(d/dt) v = u * (b1 * v + b2 * phidot + b3 * rudder)
(d/dt) phidot = u * (c1 * v + c2 * phidot + c3 * rudder)
(d/dt) phi = phidot
(d/dt) x = u * cos(phi) − v * sin(phi       (d/dt)y = u * sin(phi) + v * cos(phi)
```

The ***target angle*** **psi** is the angle between the horizontal line in Fig. 1-9a and a line joining torpedo and target. The target coordinates **xt** and **yt**, the squared distance-to-target **dd**, and the target angle **psi** are given by

```
xt = xt0 + vxt * t                  yt = yt0 + vyt * t
psi = arctan((yt − y)/(xt − x))     dd = (x − xt)² + (y − yt)²
```

We aim the torpedo at the target by making the initial value of **phi** equal to **psi**. The initial values of **u** and **v** are set to 0.

We control the rudder to keep the torpedo turned toward the target. Such simple ***pursuit guidance*** works only for low target speeds, unless you are initially more or less directly behind or in front of the moving target (Fig. 1-10). More advanced guidance systems are discussed in Ref. 14.



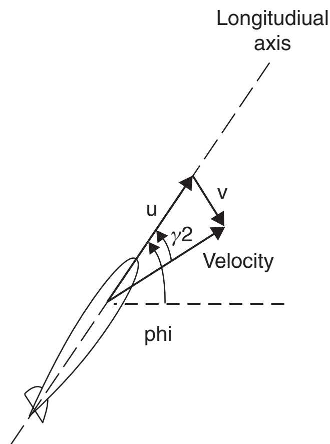


FIGURE 1-9a. Guided torpedo tracking a constant-speed target. The target angle **psi**, not shown here, is the angle between the horizontal line and the line joining the torpedo and the target.

Simple sonar guidance senses **psi** and **dd** and actuates the control-surface deflection **rudder** to implement

error = (phi − psi)   rudder = −rumax * sat(gain * error)

We increase the controller gain as the torpedo approaches the target by setting

gain = gain0 + A * t

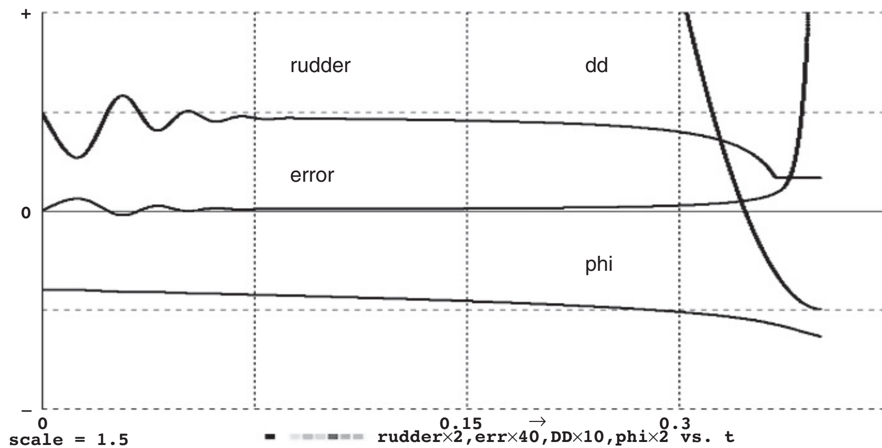


FIGURE 1-9b. Time histories of the torpedo rudder deflection, the error **phi-psi**, the angle **phi**, and the squared distance **dd** to the target (see the text).

We terminate the run when the torpedo gets close to the target, where **psi** tends to change rapidly. The second equation ensures that the absolute value of the control-surface deflection does not exceed **rumax**.

(b) Complete Torpedo-Simulation Program

Figure 1-9c lists the complete guided-torpedo program used to produce the displays in Fig. 1-9a and b. The experiment protocol first selects an integration routine, display colors, and a display scale, then sets the initial value of the integration step **DT**, the simulation runtime **TMAX**, and the number **NN** of display sampling points.

The experiment-protocol script next specifies torpedo parameters, initial target coordinates, and target-velocity components. Finally, we specify initial values for the state variables **x**, **y**, and **phi**. The initial values of the remaining state variables **u**, **v**, and **phidot** are allowed to default to zero.

The DYNAMIC program segment following the **DYNAMIC** line begins with the defined-variable assignments. We specify the target coordinates **xt** and **yt** as functions of time and then derive the target angle **psi** and the controller variables **error** and **rudder**. The DYNAMIC segment next lists the state differential equations and a ***termination command***

term rr − dd

which stops the simulation when the missile closes to within **RR = sqrt(rr)**. If it does not, our shot has failed, and the run continues to **t = TMAX**. The simulated rudder deflection **rudder** is bounded between - **rumax** and **rumax** with the limiter function **sat()** (Sec. 2-8a), which is preceded by a **step** statement to ensure correct integration (Sec. 2-11).

```
--                    GUIDED-TORPEDO SIMULATION
--     (x, y) is torpedo, (xt, yt) is target
-----------------------------------------------------------------
irule 4  |  ERMAX = 0.1 | --       variable-step RK4
display N1 | display C8  | display R  | scale = 2
DT = 0.00001  |  TMAX = 2  |  NN = 20000
-----------------------------------------------------------------
UC = 8  |  --                          torpedo parameters
a1 = 0.8155  |  a2 = 0.8155
UT = a1 * UC^2
b1 = - 15.701  |  b2 = - 0.23229  |  b3 = 0
c1 = - 303.801  |  c2 = - 44.866  |  c3 = 500
--------
gain = 300 | rumax = 0.25 | -- control parameters
RR = 0.01  |  rr = RR^2  |  --      distance to target
DD = 100 * rr
--------
vxt = 0.1  |  vyt = - 0.5  | --   target velocity vector
x = - 2  |  y = 0  |  rudder=0 | --         initial values
xt0 = 1  |  yt0 = 2
phi = atan2(yt0 - y, xt0 - x)  |  --  first aim at target
drunr
-----------------------------------------------------------------
DYNAMIC
-----------------------------------------------------------------
d/dt u = UT - a2 * u^2  |  --          state equations
d/dt v = u * (b1 * v + b2 * phidot + b3 * rudder)
d/dt phidot = u * (c1 * v + c2 * phidot + c3 * rudder)
d/dt phi = phidot
d/dt x = u * cos(phi) - v * sin(phi)
d/dt y = u * sin(phi) + v * cos(phi)
--
xt = xt0 + vxt * t  |  yt = yt0 + vyt * t  |  --     target
psi = atan2(yt - y, xt - x)  |  --            target angle
dd = (x - xt)^2 + (y - yt)^2  |  --  squared distance
----------------------
error = (phi-psi)  |  --                            control
step | --                        this is needed for sat()
rudder = - rumax * sat(gain * error)
--
term rr - dd  |  --              terminate when close
-----------------------------------------------------------------
DISPXY x, y, xt, yt | --                draw 2 xy plots
```

FIGURE 1-9c. Complete program for the guided-torpedo simulation.

Finally, the ***display command*** **DISPXY x, y, xt, yt produces** simultaneous displays of the missile and target trajectories (**y** vs. **x** and **yt** vs. **xt**). Alternative display statements can plot time histories of **phi**, **psi**, **error**, and **rudder** (Fig. 1-9b). You can load the simulation program from an editor window. Solution displays will then appear on a typed e**run** (or **zz**) command.

STOP AND LOOK

1-17. Simulation in the Real World: A Word of Caution

Simulations like our torpedo example provide insight and are nice for teaching and learning. But engineering-design simulation requires much more than solving textbook problems. In fact, the main result of a few model runs will be questions rather than answers: you will begin to see ***how much more you need to know.*** Here are just a few questions that might come up:

- Can your missile acquire the target from different directions?
- What happens if the target speed increases?
- Can you improve the design with different vehicle or control-system parameters?
- What parameter-value tolerances are acceptable?

We shall clearly always require multirun simulation studies. Figure 1-10 shows a simple example, but in practice we investigate ***combinations*** of problems like those listed. It follows that even a simple problem like our torpedo can require over a ***thousand*** simulation runs. A larger project can generate an enormous volume of

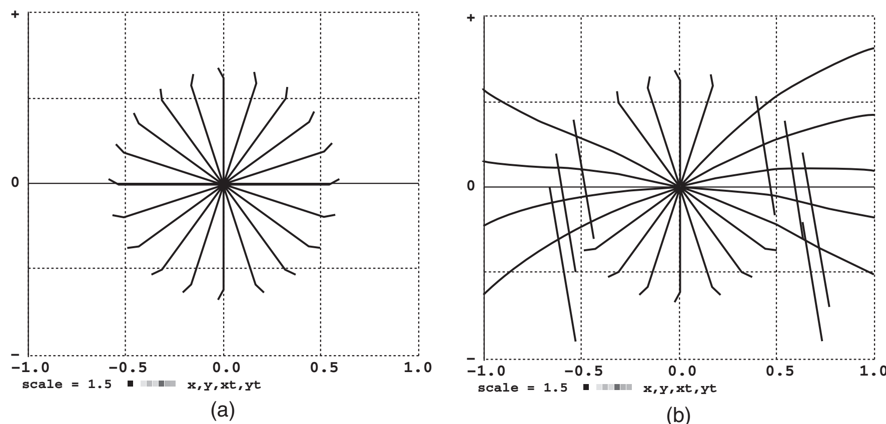


FIGURE 1-10. Multirun studies showing the results of torpedo shots at low-speed (a) and high-speed (b) targets appearing in different directions. It is a well-known fact [16,18] that the primitive pursuit-guidance scheme described in Sec. 1-17 can acquire a high-speed target only when the target track is either ahead of the missile or behind it.

simulation data. Intelligent and efficient evaluation of such results is an art rather than a science. It is our specific purpose in this book to show techniques that generate thousands of experiments in minutes and display results in various ways.

Computer simulation is convenient and dramatically cheaper than real experiments. But engineering-design models may be ***meaningless*** unless they can be validated by actual physical experiments. Very expensive prototype failures have been traced to oversimplified models (neglecting, for instance, missile fuselage bending or fuel sloshing). Simulation studies try to anticipate design problems and select test conditions that will minimize the number of expensive tests.

REFERENCES

1. Korn, G.A., and J.V. Wait: *Digital Continuous-System Simulation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
2. Korn, G.A.: *Interactive Dynamic-System Simulation*, 2nd ed., Taylor & Francis, Boca Raton, FL, 2010.

The Modelica Language

3. Tiller, M.M.: *Introduction to Physical Modeling with Modelica*, Kluwer Academic, Norwell, MA, 2004.
4. Fritzson, P.: *Principles of Object-Oriented Modeling and Simulation with Modelica*, 2nd ed., Wiley, Hoboken, NJ, 2011.
5. *DYMOLA Manual*, Dynasim A.B., Lund, Sweden, 2012.

Solution of Differential Equations and Differential-Algebraic Equations

6. Cellier, F.: *Numerical Simulation of Dynamic Systems*, Springer-Verlag, New York, 2010.
7. Cellier, F., and E. Kofman: *Continuous-System Simulation*, Springer-Verlag, New York, 2006.
8. Gear, C.W.: DIFSUB, Algorithm 407, *Communications of the ACM*, **14**(3), 3–7, 1971.
9. Asher, U.M., and L. Petzold: *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM Press, Philadelhia, 1998.
10. Petzold, L.: A Description of DASSL, a Differential-Algebraic-Equation Solver, in *Scientific Computing*, Stepleman, R.S. (ed.), North-Holland, Amsterdam, 1989.
11. Stoer, J., et al.: *Introduction to Numerical Analysis*, Springer-Verlag, New York, 2002.

Missile Guidance

12. Howe, R.M: in *Hybrid Computation*, Karplus, W.J., and G.A. Bekey (eds.), Wiley, New York, 1968.

JWBS097-c01 JWBS097-Korn Printer: Yet to Come January 16, 2013 8:6 6.125in×9.25in

13. Siouris, G.M: *Missile Guidance and Control*, Springer-Verlag, New York, 2003.
14. Thomson-Smith, L.D.: *Guided Missiles: Modern Precision Weapons*, Fastbook Publishing, 2008.
15. Yanushevsky, R.: *Modern Guided Missiles*, CRC/Taylor & Francis, Boca Raton, FL, 2008.