

 WILEY

ADVANCED DYNAMIC-SYSTEM SIMULATION

MODEL-REPLICATION TECHNIQUES AND MONTE CARLO SIMULATION

GRANINO A. KORN



Includes
CD-ROM

Advanced Dynamic-system Simulation

Model-replication Techniques and Monte Carlo Simulation

Granino A. Korn

University of Arizona
Tucson, Arizona



WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

Advanced Dynamic-system Simulation



THE WILEY BICENTENNIAL—KNOWLEDGE FOR GENERATIONS

Each generation has its unique needs and aspirations. When Charles Wiley first opened his small printing shop in lower Manhattan in 1807, it was a generation of boundless potential searching for an identity. And we were there, helping to define a new American literary tradition. Over half a century later, in the midst of the Second Industrial Revolution, it was a generation focused on building the future. Once again, we were there, supplying the critical scientific, technical, and engineering knowledge that helped frame the world. Throughout the 20th Century, and into the new millennium, nations began to reach out beyond their own borders and a new international community was born. Wiley was there, expanding its operations around the world to enable a global exchange of ideas, opinions, and know-how.

For 200 years, Wiley has been an integral part of each generation's journey, enabling the flow of information and understanding necessary to meet their needs and fulfill their aspirations. Today, bold new technologies are changing the way we live and learn. Wiley will be there, providing you the must-have knowledge you need to imagine new worlds, new possibilities, and new opportunities.

Generations come and go, but you can always count on Wiley to provide you the knowledge you need, when and where you need it!

WILLIAM J. PESCE
PRESIDENT AND CHIEF EXECUTIVE OFFICER

PETER BOOTH WILEY
CHAIRMAN OF THE BOARD

Advanced Dynamic-system Simulation

Model-replication Techniques and Monte Carlo Simulation

Granino A. Korn
University of Arizona
Tucson, Arizona



WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2007 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400, fax 978-750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, 201-748-6011, fax 201-748-6008, or online at <http://www.wiley.com/go/permission>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at 877-762-2974, outside the United States at 317-572-3993 or fax 317-572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Korn, Granino Arthur, 1922-

Advanced dynamic-system simulation : model-replication techniques and Monte Carlo simulation / by Granino A. Korn.

p. cm.

Includes index.

ISBN 978-0-470-08188-4 (cloth/cd)

1. System analysis--Simulation methods. 2. Monte Carlo method. I. Title.

QA402. K665 2007

003' .85-- dc22

200601618

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

Preface	xiii
Chapter 1. Introduction to Dynamic-system Simulation	1
DYNAMIC-SYSTEM MODELS AND COMPUTER PROGRAMS	1
1-1. Computer Modeling and Simulation	1
1-2. Differential-equation Models	2
1-3. Interactive Modeling—Experiment Protocol and Simulation Studies	3
1-4. Simulation Software	4
1-5. OPEN DESIRE and DESIRE	4
HOW A SIMULATION RUN WORKS	5
1-6. Sampling the DYNAMIC Segment Variables	5
1-7. Numerical Integration	10
(a) Euler Integration	10
(b) Improved Integration Rules	10
1-8. Sampling Times and Integration Steps	11
1-9. Sorting Defined-variable Assignments	12
EXAMPLES OF SIMPLE APPLICATIONS	12
1-10. Oscillators and Computer Displays	12
(a) A Linear Harmonic Oscillator	12
(b) A Nonlinear Oscillator and Duffing's Differential Equation	15
1-11. Space Vehicle Orbits—Variable-step Integration	15
1-12. A Population-dynamics Model	18
1-13. Splicing Multiple Simulation Runs: Billiard-ball Simulation	20

vi Contents

CONTROL-SYSTEM EXAMPLES	22
1-14. An Electrical Servomechanism with Motor Field Delay and Saturation	22
1-15. Control-system Frequency Response	24
1-16. Simulation of a Simple Guided Missile	25
(a) A Guided Torpedo	25
(b) The Complete Simulation Program	28
WHAT DO WE DO WITH ALL THIS?	29
1-17. Simulation Studies in the Real World: A Word of Caution	29
REFERENCES	30
 Chapter 2. Models with Difference Equations, Limiters, and Switches	 32
SAMPLED-DATA ASSIGNMENTS AND DIFFERENCE EQUATIONS	32
2-1. Sampled-data Difference Equation Systems	32
2-2. “Incremental” Form of Simple Difference Equations	34
2-3. Combining Differential Equations and Sampled-data Operations	35
2-4. A Simple Example	36
2-5. Initializing and Resetting Sampled-data Variables	38
EXAMPLES OF MIXED CONTINUOUS/SAMPLED-DATA SYSTEMS	38
2-6. The Guided Torpedo with Digital Control	38
2-7. Simulation of a Plant with a Digital PID Controller	40
MODELING LIMITERS AND SWITCHES	42
2-8. Limiters, Switches, and Comparators	42
(a) Limiter Functions	42
(b) Switching Functions and Comparators	42
2-9. Numerical Integration of Switch and Limiter Outputs, Event Prediction, and Display Problems	45
2-10. Using Sampled-data Assignments	46
2-11. Using the step Operator and Heuristic Integration-step Control	46
2-12. Example: Simulation of a Bang-bang Servomechanism	47
LIMITERS, SWITCHES, AND DIFFERENCE EQUATIONS	49
2-13. Limiters, Absolute Value, and Maximum/Minimum Selection	49
2-14. Output-limited Integration	50
2-15. Modeling Signal Quantization	50
2-16. Continuous-variable Difference Equations with Switching and Limiter Operations	51
(a) Introduction	51
(b) Track-hold Simulation	52
(c) Maximum- and Minimum-value Holding	53

	Contents	vii
(d) Simple Backlash and Hysteresis Models		53
(e) The Comparator with Hysteresis (Schmitt Trigger)		54
2-17. Signal Generators and Signal Modulation		56
REFERENCES		58
Chapter 3. Programs with Vector/Matrix Operations and Submodels		59
VECTOR ASSIGNMENTS AND VECTOR DIFFERENTIAL EQUATIONS		59
3-1. Arrays, Subscripted Variables, and State-variable Declarations		59
3-2. Vector Operations in DYNAMIC Program Segments— The Vectorizing Compiler		60
(a) Vector Assignments and Vector Expressions		60
(b) Vector Differential Equations		61
(c) Vectorization and Model Replication—Significant Applications		62
3-3. Matrix-vector Products in Vector Expressions		63
(a) Definition		63
(b) A Simple Example: Resonating Oscillators		64
3-4. Vector Sampled-data Assignments and Vector Difference Equations		64
3-5. Sorting Vector and Subscripted-variable Assignments		66
MORE VECTOR OPERATIONS		66
3-6. Index-shifted Vectors		66
3-7. Sums, DOT Products, and Vector Norms		67
(a) Sums and DOT Products		67
(b) Euclidean, Taxicab, and Hamming Norms		67
3-8. Maximum/Minimum Selection and Masking		68
(a) Maximum/Minimum Selection		68
(b) Masking Vector Expressions		69
MATRIX OPERATIONS		69
3-9. Matrix Operations in Experiment-protocol Scripts		69
3-10. Matrix Assignments and Difference Equations in DYNAMIC Program Segments		70
3-11. Vector and Matrix Operations using Equivalent Vectors		71
VECTORS IN PHYSICS AND CONTROL-SYSTEM PROBLEMS		71
3-12. Vectors in Physics Problems		71
3-13. Simulation of a Nuclear Reactor		72
3-14. Linear Transformations and Rotation Matrices		72
3-15. State-equation Models for Linear Control Systems		74
USER-DEFINED FUNCTIONS AND SUBMODELS		75
3-16. User-defined Functions		75

3-17. Submodels	76
(a) Submodel Declaration and Invocation	76
(b) Submodels with Differential Equations	78
3-18. Dealing with Sampled-data Assignments, Limiters, and Switches	78
REFERENCES	79

Chapter 4. Parameter-influence Studies, Model Replication, and Monte Carlo Simulation 80

PARAMETER-INFLUENCE STUDIES AND VECTORIZATION	80
4-1. Exploring the Effects of Parameter Changes	80
4-2. Repeated Runs and Model-Replication (Vectorization)	81
(a) A Simple Repeated-run Study	81
(b) Model Replication	82
(c) Dealing with Multiple Parameters	84
4-3. Programming Parameter-influence Studies	85
(a) Introduction	85
(b) Measures of System Effectiveness	85
(c) Crossplotting Results	86
(d) Maximum/Minimum Selection	87
(e) Iterative Parameter Optimization	87
RANDOM PROCESSES AND RANDOM PARAMETERS	88
4-4. Random Processes and Monte Carlo Simulation	88
4-5. Generating Random Parameters and Random Initial Values	89
MONTE CARLO SIMULATION OF DYNAMIC SYSTEMS	89
4-6. Repeated-run Monte Carlo Simulation	89
(a) Taking Statistics on Repeated Simulation Runs	89
(b) Sequential Monte Carlo Studies	91
(c) Example: Effects of Gun-elevation Errors on the 1776 Cannon	91
4-7. Vectorized (Model-replicating) Monte Carlo Simulation	93
(a) Vectorized Monte Carlo Study of the 1776 Cannon Shot	93
(b) Interactive Monte Carlo Simulation: Computing Time Histories of Statistics with Compiled DOT Operations	96
4-8. Statistical Relative Frequencies, Sample Ranges, and Other Statistics	96
4-9. Post-run Probability-density Estimation	97
(a) A Simple Probability-density Estimate	97
(b) Triangle and Parzen Windows	98
(c) Computation and Display of Parzen Window Estimates	99
4-10. Combining Vectorized and Repeated-run Monte Carlo Simulation	100
REFERENCES	103

Chapter 5. Random-process Simulation and Monte Carlo Studies with Noisy Signals	105
COMPUTER MODELS OF NOISE PROCESSES	105
5-1. Noise in DYNAMIC Program Segments	105
5-2. Sampled-data Random Processes	105
(a) A Platform for Sampled-data Experiments	105
(b) A Sampled-data Random Process Model: Coin Tossing	106
(c) Recursive Sampled-data Addition and Time Averaging	106
5-3. Modeling Continuous Noise	107
(a) Deriving “Continuous” Noise from Periodic Pseudorandom Samples	107
(b) “Continuous” Time Averages	109
5-4. Problems with Simulated Noise	109
MONTE CARLO SIMULATION WITH NOISY SIGNALS	109
5-5. Gambling Returns	109
5-6. A Continuous Random Walk	112
5-7. The 1776 Cannonball with Air Turbulence	113
SIMULATION OF NOISY CONTROL SYSTEMS	116
5-8. Monte Carlo Simulation of a Nonlinear Servomechanism: A Noise-input Test	116
5-9. Monte Carlo Study of Control-system Errors Caused by Noise	119
ADDITIONAL TOPICS	119
5-10. Monte Carlo Optimization	119
5-11. A Convenient Heuristic Method for Testing Pseudorandom Noise	121
5-12. An Alternative to Monte Carlo Simulation	121
(a) Introduction	121
(b) Dynamic Systems with Random Perturbations	122
(c) Mean Square Errors in Linearized Systems	122
REFERENCES	123
 Chapter 6. Vector Models of Neural Networks	 125
NEURAL-NETWORK SIMULATION	125
6-1. Neural-network Models and Pattern Vectors	125
6-2. Simple Vector Operations Model Neural-network Layers	126
6-3. Normalizing and Contrast-enhancing Neuron Layers	127
6-4. Multilayer Networks	128
6-5. Exercising a Neural-network Model	129
(a) Computing Successive Neuron Layer Outputs	129
(b) Using Pattern-row Matrices	129
(c) Pattern Input from Files	130

REGRESSION AND PATTERN CLASSIFICATION	130
6-6. Mean-square Regression	131
6-7. Pattern Classification	131
NEURAL-NETWORK TRAINING: PATTERN CLASSIFICATION AND ASSOCIATIVE MEMORY	132
6-8. Linear Pattern Classifiers	132
6-9. The LMS Algorithm	132
6-10. A Softmax Image Classifier	133
(a) Problem Statement and Experiment-protocol Script	133
(b) Network Model and Training	134
(c) Test Runs and A Posteriori Probabilities	137
6-11. Associative Memory	138
NONLINEAR MULTILAYER NETWORKS	138
6-12. Backpropagation Networks	138
(a) The Backpropagation Algorithm	138
(b) Discussion	140
(c) Examples and Neural-network Submodels	141
6-13. Radial-basis-function Networks	141
(a) Basis-function Expansion and Linear Optimization	141
(b) Radial Basis Functions	144
COMPETITIVE-LAYER PATTERN CLASSIFICATION	146
6-14. Template-pattern Matching	146
6-15. Unsupervised Pattern Classifiers	147
(a) Simple Competitive Learning	147
(b) Learning with Conscience	148
6-16. Experiments with Pattern Classification and Vector Quantization	149
(a) Pattern Classification	149
(b) Vector Quantization	150
6-17. Simplified Adaptive-resonance Emulation	151
6-18. Biologically Plausible Competition: Correlation Matching	153
SUPERVISED COMPETITIVE LEARNING	154
6-19. Supervised Competitive Classifiers: The LVQ Algorithm	154
6-20. Counterpropagation Networks	155
NEURAL NETWORKS WITH MEMORY	155
6-21. Neural Networks and Memory	155
6-22. Networks with a Delay-line Input Layer	157
(a) Vector Model of a Tapped Delay Line	157
(b) Simple Linear Filters	158

(c) Linear Matched Filters, Signal Classifiers, and Model Matching	159
(d) A Nonlinear Predictor Trained with Backpropagation	159
6-23. The Gamma Delay Line Layer	162
PULSED-NEURON REPLICATION	163
6-24. Pulsed-neuron Models	163
6-25. A Simple Integrate and Fire Model	164
6-26. Neuron-model Replication	166
REFERENCES	168
Chapter 7. More Applications of Vector Models	171
A VECTORIZED SIMULATION WITH LOGARITHMIC PLOTS	171
7-1. The EUROSIM No. 1 Benchmark Problem	171
7-2. Vectorized Simulation with Logarithmic Plots	171
MODELING FUZZY-LOGIC FUNCTION GENERATORS	172
7-3. Rule Tables Specify Heuristic Functions	172
7-4. Fuzzy-set Logic	174
(a) Fuzzy Sets and Membership Functions	174
(b) Fuzzy Intersections and Unions	175
(c) Joint Membership Functions	175
(d) Normalized Fuzzy-set Partitions	175
7-5. Fuzzy-set Rule Tables and Function Generators	178
7-6. Simplified Function Generation with Fuzzy Basis Functions	179
7-7. Vector Models of Fuzzy-set Partitions	179
(a) Gaussian Bumps—Effects of Normalization	179
(b) Triangle Functions	180
(c) Smooth Fuzzy Basis Functions	181
7-8. Vector Models for Multidimensional Fuzzy-set Partitions	181
7-9. Example: Fuzzy-logic Control of a Servomechanism	182
(a) Problem Statement	182
(b) Experiment Protocol and Rule Table	183
(c) DYNAMIC Program Segment and Results	184
PARTIAL DIFFERENTIAL EQUATIONS	186
7-10. The Method of Lines	186
7-11. The Vectorized Method of Lines	188
(a) Introduction	188
(b) Using Differentiation Operators	188
(c) Numerical Problems	191
7-12. The Heat-conduction Equation in Cylindrical Coordinates	192

xii Contents

7-13. Generalizations	192
7-14. A Simple Heat-exchanger Model	194
REPLICATION OF AGROECOLOGICAL MODELS ON MAP GRIDS	197
7-15. A Geographical Information System	197
7-16. Modeling the Evolution of Landscape Features	197
REFERENCES	199
Appendix	201
ADDITIONAL REFERENCE MATERIAL	201
A-1. Example of a Radial-basis-function Network	201
A-2. A Fuzzy-basis-function Network	203
A-3. The CLEARN Algorithm	205
REFERENCES	206
PROGRAMS IN THE BOOK CD	210
STREAMLINED OPERATION OF DESIRE PROJECTS UNDER LINUX	210
Index	213

Preface

Simulation is experimentation with models. This book describes new computer programs for interactive modeling and simulation of dynamic systems, such as aerospace vehicles, control systems, and biological systems. Simulation studies for design or research can involve many hundreds of model changes, so programming must be convenient, and computations must be as fast as possible.

This book is about advanced simulation programming and describes many new techniques. We provide only a brief review of routine simulation programming but demonstrate computer software for remarkably fast and respectably large simulation studies on inexpensive personal computers or workstations. For hands-on experiments, the enclosed CD contains an industrial-strength software package rather than a toy demonstration program.¹

¹OPEN DESIRE solves up to 40,000 ordinary differential equations under Linux, and up to 20,000 differential equations under Microsoft WindowsTM, so that one can implement respectable vectorized Monte Carlo studies. The DESIRE language, widely used since 1985, accepts scalar and vector differential equations and difference equations in a natural mathematical notation, for example,

$$\begin{aligned}d/dt \mathbf{x} &= -\mathbf{x} * \cos(\mathbf{w} * t) + 2.22 * \mathbf{a} * \mathbf{x} \\ \text{Vector } \mathbf{y} &= \mathbf{A} * \mathbf{x} + \mathbf{B} * \mathbf{u}\end{aligned}$$

Programs entered in editor windows immediately compile, execute, and produce solution displays. The program in the book CD allows the user to experiment with all the examples in the text. The Open Source programs in the book CD include binary and source code and a comprehensive reference manual. The Linux version can be recompiled for other Unix-type systems, including Solaris[©] and Cygwin (Unix under Windows[©]).

The included OPEN DESIRE program for Linux solves up to 40,000 ordinary differential equations and implements exceptionally fast and convenient vector operations. A smaller educational 20,000 differential-equations version for Microsoft WindowsTM can be obtained without charge from the author by sending an email to gatmkorn@aol.com. The user can run, edit, and modify the example simulations keyed to the figures in this book, plus many other examples. Many of our programming principles also apply to simulation programs other than DESIRE.

Chapter 1 introduces our subject with a few programs for small differential-equation models, including a simple guided-missile simulation. The remainder of the book presents new material. Chapter 2 begins with a careful discussion of models that involve sampled-data operations and sampled-data difference equations together with differential equations. We model mixed analog–digital systems such as simulated digital controllers and systems with limiters and switches. At this point, we show that many very useful devices (e.g., track-hold circuits, trigger circuits, signal generators, automatic scaling) are neatly and efficiently modeled with simple difference equations. Last, but not least, we propose improved techniques for proper numerical integration of switched variables.

Truly powerful simulation programs need a readable notation for vector and matrix assignments, differential equations, and difference equations. Chapter 3 introduces a novel vector compiler that produces very fast programs for vector and matrix operations. We also demonstrate efficient use of submodels. We present examples from control engineering and nuclear-reactor simulation. The following chapter then shows how we use vectors to replicate complete models.

Chapter 4 describes practical model replication (vectorization): a single simulation run with a vector model will replace hundreds or thousands of conventional simulation runs. We apply the vectorizing compiler introduced in Chapter 3 to parameter-influence studies and Monte Carlo simulation of dynamic systems with noise-perturbed parameters and random initial conditions. We compare repeated-run and vectorized Monte Carlo studies of a weapon trajectory. Our interactive programs produce not just time histories of system variables but also time histories of statistics such as averages, mean squares, and probability estimates. We also show explicitly how to estimate probability densities.

Chapter 5 discusses more difficult vectorized Monte Carlo simulations involving time-varying noise, which has to be derived from periodic pseudo-random noise samples. Examples include Monte Carlo simulation of a continuous random walk, another trajectory study, and two vectorized control-system simulations. An inexpensive 2.4-GHz personal computer exercised 1000 random-input control-system models in seconds. We also describe a new heuristic test for the quality of pseudorandom noise.

Chapter 6 discusses vector models of neural networks including a new pulsed-neuron model. Chapter 7 deals with vectorized programs for fuzzy-set controllers, partial differential equations, and agroecological models replicated at many points of a landscape map. The Appendix gets a small selection of reference material out of the way of the main text.

GRANINO A. KORN
Chelan, Washington
October 2006

1

Introduction to Dynamic-system Simulation

DYNAMIC-SYSTEM MODELS AND COMPUTER PROGRAMS

1-1. Computer Modeling and Simulation

Simulation is *experimentation with models*. Simulation for engineering design, research, and education studies must rapidly exercise a wide variety of models and then store and access a large volume of results. This is practical only with models programmed on computers.

Dynamic-system models relate model-system states to earlier states. Classical physics, for example, predicts continuous changes of quantities such as position, velocity, or voltage with continuous time. Computer simulation of such systems started in the aerospace industry and is now indispensable in biology, medicine, and agroecology as well as in all engineering disciplines. At the same time, discrete-event simulation has gained importance for business and military planning.

Simulation is at its best when combined with mathematical analyses. But simulation results can often provide insight and possibly useful decisions where exact analysis is impractical. This was true for many early control-system optimizations. As another example, Monte Carlo simulations simply measure statistics over repeated experiments to solve problems that are too complicated for probability theory analysis. Simulation results must eventually be validated by real experiments, just like analytical results.

2 Introduction to Dynamic-system Simulation

Computer simulations can be speeded up or slowed down at the experimenter's convenience. You 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”: you can “fly” a real autopilot—or a real human pilot—on a tilting platform controlled by a computer flight simulation. In this book, we are interested in very fast simulation, for we want to study the effects of many different model changes. Specifically, we want to

1. Enter and edit programs in convenient editor windows.
2. Use typed or graphical-interface commands to start, stop, and pause simulations, to select displays, and to make parameter changes. Results ought to appear immediately to provide an intuitive “feel” for the effects of model changes (interactive modeling).
3. Program systematic parameter-influence studies and produce cross-plots or tables.
4. Program model changes to optimize effectiveness measures, and study effects of random parameter changes or random model inputs by taking statistics on repeated simulations (Monte Carlo simulation).

1-2. Differential-equation Models

Continuous-system simulation models delayed interactions of physical state variables $\mathbf{x1}$, $\mathbf{x2}$, ... with first-order ordinary differential equations (state equations)

$$(d/dt) \mathbf{x_i} = \mathbf{f_i(t; x1, x2, \dots; y1, y2, \dots; a1, a2, \dots)} \quad (i = 1, 2, \dots) \quad (1-1a)$$

Here \mathbf{t} represents the time, and the quantities

$$\mathbf{y_j} = \mathbf{g_j(t; x1, x2, \dots; y1, y2, \dots; b1, b2, \dots)} \quad (j = 1, 2, \dots) \quad (1-1b)$$

are defined variables. $\mathbf{a1}$, $\mathbf{a2}$, ..., and $\mathbf{b1}$, $\mathbf{b2}$, ... are constant model parameters.

Simulation programs exercise such models by solving the state-equation system (1-1) to produce time histories of the system variables $\mathbf{x_i = x_i(t)}$ and $\mathbf{y_j = y_j(t)}$ for $\mathbf{t = t_0}$ to $\mathbf{t = t_0 + TMAX}$, starting with given initial values $\mathbf{t_0}$ and $\mathbf{x_i(t_0)}$. In Section 1-6 and Chapter 2, we shall add sampled-data operations representing periodic inputs and outputs, sample-holds, and digital controllers.

The state variables $\mathbf{x_i}$ are system outputs. They start at $\mathbf{t = t_0}$ with given initial values; subsequent values are produced by an integration routine (Section 1-7) from the $\mathbf{f_i}$ -values generated by the preceding execution (derivative call) of the operations (1).

There are three kinds of defined variables $\mathbf{y_j}$:

1. system inputs (specified functions of the time \mathbf{t})
2. system outputs
3. intermediate results needed to compute the derivatives $\mathbf{f_i}$

It must be possible to sort the defined-variable assignments (1-1b) into a procedure that successively derives all the $\mathbf{y_j}$ from state variables $\mathbf{x_i}$ and/or the time \mathbf{t} without recurrence relations or “algebraic loops” (Section 1-9).

Some dynamic systems (e.g., systems involving interconnected mechanical devices in automotive engineering and robotics) are modeled with differential-equation systems that cannot be explicitly solved for state-variable derivatives as in Eq. (1-1). Simulation then requires solution of algebraic equations at each integration step. Such differential-algebraic-equation (DAE) systems are not treated in this book. References [1–4] describe suitable mathematical methods and special software.

1-3. Interactive Modeling-Experiment Protocol and Simulation Studies

Practical computer simulation is not simply a matter of programming and solving model equations. We must also make it convenient to modify our models and try many different experiments (see also Section 1-5). In addition to DYNAMIC program segments listing the model equations (1-1), each simulation study requires an experiment protocol program that sets and changes initial conditions and model parameters, calls computer runs, and displays or tabulates solutions for different model configurations.

The simplest experiment protocols are just sequences of successive commands, say

a = 20.0		b = - 3.35	<i>(set parameter values)</i>
x = 12.0			<i>(set the initial value of \mathbf{x})</i>
drun			<i>(make a differential-equation -solving simulation run)</i>
reset			<i>(reset initial values)</i>
a = 20.1			<i>(change model parameters)</i>
b = b - 2.2			
drun			<i>(try another run)</i>

Each **drun** command calls a differential-equation-solving simulation run, and **reset** resets initial conditions. Typed commands ought to execute immediately to permit interactive modeling. The operator inspects the solution output after each simulation run and then types new commands for the next run. Command-mode operation also permits interactive program debugging [5].

A simulation study combines such commands into a storable program segment (experiment-protocol script) that can branch and loop to call repeated simulation runs for different parameter combinations. Simulation studies may involve thousands of model and parameter changes, so programming must be easy and computations must be as fast as possible. This is why we like to interpret experiment-protocol scripts and compile the program segments executing the actual simulation runs.

1-4. Simulation Software

Commercially available equation-oriented simulation programs such as ACSLTM accept system equations in a more or less human-readable form, sort defined variable assignments as needed, and feed the sorted equations to an optimizing Fortran or c compiler [5]. Berkeley Madonna and DESIRE (see below) have built-in equation-language compilers and execute immediately. Block-diagram interpreters (e.g., SimulinkTM, VissimTM, and the open-source program Scicos) permit graphical block-diagram composition and immediately execute interpreted simulation runs. Such programs usually provide equation-language blocks for complicated expressions. Interpreted code is slow; production runs are sometimes translated into c for faster execution. Alternatively, ACSLTM, Easy5TM, and Berkeley Madonna have block-diagram preprocessors for compiled simulation programs. More advanced modeling is possible with the Modelica language [6–8].

1-5. OPEN DESIRE and DESIRE

The simulation programs described in this book, and, in particular, our new techniques for model replication (vectorization), Monte Carlo simulation, and submodels (Chapters 3–7), use the open-software simulation package OPEN DESIRE for Linux, Unix including Cygwin (Unix under Windows), and Microsoft WindowsTM, or the commercially available DESIRE/2000 program for Windows.¹ DESIRE simulation systems allow inexpensive personal computers and workstations solve thousands of differential equations in seconds.

¹ The earlier (1995) Windows version of DESIRE discussed in References [1,2] lacks the vector-compilation features used in this book.

DESIRE uses double-precision (64-bit) floating-point arithmetic and accepts command scripts and model descriptions in a readable mathematical notation such as

$$y = a * \cos(x) + b$$

$$d/dt x = -x + 4 * y$$

Command scripts can include operating-system calls, shell scripts, and calls to other computer programs. DESIRE's command-script language is itself a general-purpose mathematical language and handles vectors, matrices, and even complex numbers (e.g., for frequency-response and root-locus plots) [9]. Programs are entered and edited in editor windows (Fig. 1-1). Each program begins with an experiment-protocol script that is interpreted much like an advanced Basic dialect. When the experiment-protocol script encounters a **drun** statement, a built-in runtime compiler automatically compiles a DYNAMIC program segment listing model equations. The state-equation-solving simulation run then executes at once and produces solution displays in bright color.

Very fast compilation (typically under 50 ms) simplifies interactive modeling. Experimenters can immediately observe results of programmed or screen-edited models and experiment-protocol changes. One can enter and edit different models in multiple editor windows and run these models in turn to compare results (Fig. 1-1). Runtime displays show solution time histories and error messages during rather than after each simulation run, so that you can save time by aborting undesirable runs before they complete.

The experiment-protocol script starting each DESIRE program defines an experiment. Subsequent DYNAMIC program segments define models used in the experiment and specify runtime input/output requests. An experiment protocol can call multiple DYNAMIC segments with different models, different versions of the same model, and/or different input/output operations.

HOW A SIMULATION RUN WORKS

1-6. Sampling the DYNAMIC Segment Variables

When **drun** calls a simulation run, the program initializes input/output operations requested by the DYNAMIC program segment. The independent variable **t** (simulation time) and the differential-equation state variables start with initial values assigned by the experiment protocol.² A first pass through the DYNAMIC-segment code [Eq. (1-1)] produces initial values of the defined

² Unspecified initial values of unsubscripted differential-equation state variables conveniently default to 0.

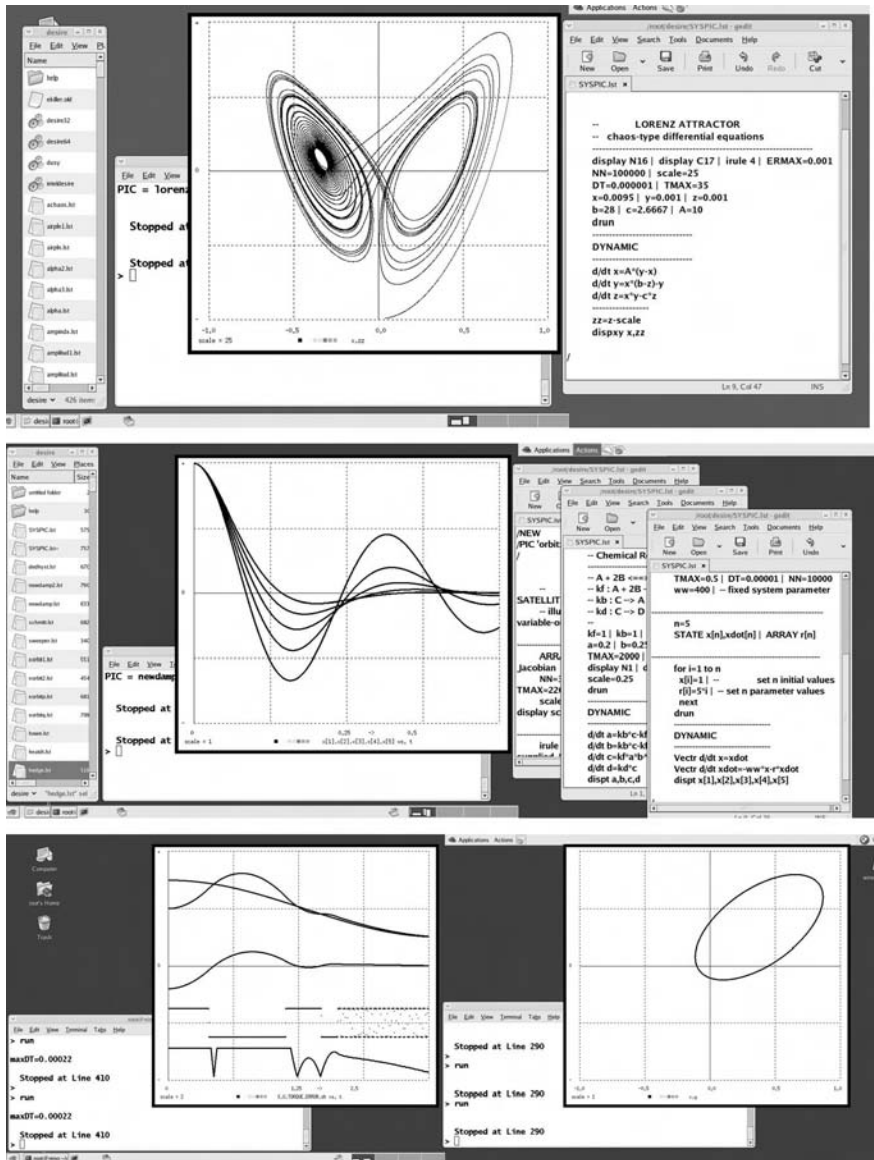


FIGURE 1-1a. OPEN DESIRE running under Linux. The first dual-monitor screen shows a file-manager window for calling user programs, the DESIRE command window, a graph window, and an editor window. The second screen has a file manager and three editor windows; one can click on any file-manager or editor window to run and compare different programs. On the third screen, two independent simulations run in separate command windows. Solution graphs were set to black-on-white for publication, but normally each displayed curve has a different color.

variables [Eq. (1-1b)]. Unless stopped, simulations run from the initial time $t = t_0$ to $t = t_0 + TMAX$. You can stop a simulation run by typing **ctrl c** and **space** (**zz** under Windows), and restart or extend a run with **drun**.

DESIRE normally samples DYNAMIC-segment variables for output (usually to displays) or sampled-data operations at **NN** uniformly spaced sampling times (communication times):

$$\begin{aligned} t = t_0, t_0 + COMINT, t_0 + 2 \text{ } COMINT, \dots, \\ t_0 + (NN - 1)COMINT = t_0 + TMAX \end{aligned} \quad (1-2a)$$

with

$$COMINT = TMAX/(NN - 1) \quad (1-2b)$$

The experiment-protocol script sets appropriate values of t_0 , **TMAX**, and **NN** or uses default values listed in the DESIRE manual.

If the DYNAMIC program segment contains differential equations (d/dt or **Vectr d/dt** statements), then t_0 defaults to $t_0 = 0$ unless another value is specified. Starting at $t = t_0$, the integration routine increments t by successive constant or variable **DT** steps until t reaches the next data-sampling communication point (Fig. 1-2a). Within each integration step, numerical integration

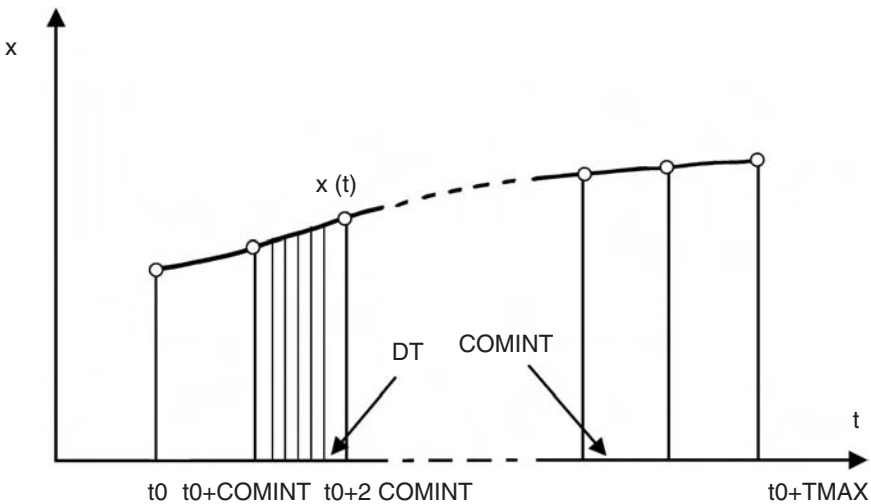


FIGURE 1-2a. Time history of a simulation variable, showing sampling times $t = t_0, t_0 + COMINT, t_0 + 2 \text{ } COMINT, \dots, t_0 + TMAX$ and some integration steps. In the figure, all integration steps end on a sampling point. This is always true for variable-step integration rules, but fixed integration steps **DT** may overshoot the sampling points by a small fraction of **DT**, as shown in Figure 1-2b.

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.

approximates continuous updating of the “continuous” model variables **t**, **xi**, and **yj**. Each integration step usually requires more than one derivative call executing the model equations (1-1) (Section 1-7; [10–18]).

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 sampled-data assignments and execute at successive communication times [Eq. (1-2)], except for assignments preceded by a **SAMPLE m** statement, where **m** is an integer >1. Such assignments execute only at **t = t0** and then at every **mth** communication point. This permits multirate sampling. DESIRE admits only one **SAMPLE m** statement per DYNAMIC program segment.

Differential-equation-solving DYNAMIC segments can also include sampled-data assignments that execute only at the periodic sampling points (1-2). Such assignments model sampled-data controllers and noise generators and must be collected in sections following an **OUT** and/or **SAMPLE m** statement at the end of the DYNAMIC program segment (Section 2-3).

DYNAMIC-segment input/output (e.g., 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 this case, input/output occurs at **t = t0**, then at every **MMth** 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 (Section 5-4) and/or for sampling switch and limiter functions (Sections 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. The simulation will run faster if such assignments are programmed as sampled-data operations following an **OUT** statement.

1-7. Numerical Integration

(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 (see also Appendix)

$$\begin{aligned} \mathbf{x}_i(\mathbf{t} + \mathbf{DT}) &= \mathbf{x}_i(\mathbf{t}) + \mathbf{f}_i[\mathbf{t}; \mathbf{x}_1(\mathbf{t}), \mathbf{x}_2(\mathbf{t}), \dots; \mathbf{y}_1(\mathbf{t}), \mathbf{y}_2(\mathbf{t}), \dots] \mathbf{DT} \\ (i &= 1, 2, \dots, n) \end{aligned} \quad (1-3)$$

where \mathbf{f}_i is the value of \mathbf{dx}/\mathbf{dt} calculated by the derivative call executing Eq. (1-1) at the time \mathbf{t} .

The integration routine loops until \mathbf{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 $\mathbf{t} = \mathbf{t}_0 + \mathbf{TMAX}$ unless the run is stopped either by the user or by a programmed termination (**term**) statement.

(b) Improved Integration Rules

The Euler integration rule [Eq. (1-3)] simply increments each state variable by an amount proportional to its last computed derivative. This does not approximate true integration well except for very small integration steps \mathbf{DT} . Improved updating requires multiple derivative calls per integration step \mathbf{DT} [10–18]. This can actually 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 values of the \mathbf{x}_i as polynomials based on values of the \mathbf{x}_i and \mathbf{f}_i at several past times $\mathbf{t} - \mathbf{DT}, \mathbf{t} - 2\mathbf{DT}, \dots$.
- *Runge-Kutta rules* precompute two or more approximate derivative values in the interval $(\mathbf{t}, \mathbf{t} + \mathbf{DT})$ 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,

$$\begin{aligned} \mathbf{x}_i(\mathbf{t} + \mathbf{DT}) = \mathbf{x}_i(\mathbf{t}) + \mathbf{f}_i[\mathbf{t} + \mathbf{DT}; \mathbf{x}_1(\mathbf{t} + \mathbf{DT}), \mathbf{x}_2(\mathbf{t} + \mathbf{DT}), \dots; \\ \mathbf{y}_1(\mathbf{t} + \mathbf{DT}), \mathbf{y}_2(\mathbf{t} + \mathbf{DT}), \dots] \mathbf{DT} \quad (\mathbf{i} = 1, 2, \dots, \mathbf{n}) \end{aligned} \quad (1-4)$$

require a program that solves the predictor equation (1-4) for **xi(t + DT)** at each step. Implicit rules clearly involve more computation, but they may admit larger **DT** values without numerical instability.

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, 7-7, and 7-8 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. Sections 2-10 to 2-12 discuss this problem 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 runtime **TMAX** and the number of samples **NN** needed for display, listings, and/or sampled-data models. DESIRE returns an error message if an integration-step value **DT** larger than **COMINT = TMAX/(NN - 1)** is selected, for the program must never sample data within integration steps. 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 Sections 2-9 to 2-11).

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 error message warns if the initial **DT** value exceeds **COMINT**. Fixed-step integration routines, however, 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 Eq. (1-2b). This does not cause errors in displays or listings, for each **x(t)** value is still associated with its correct **t** value. But if output listings at specified periodic sampling times (1-2) are needed, one 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 derivative call of the differential-equation-solving integration routine. Each derivative or defined-variable assignment uses the value of **t** and the values of the state variables **xi** computed by the last derivative call. Derivative and defined-variable values for **t = t0** are derived from the initial state-variable values and **t0** by an extra initial derivative call.

The defined-variable assignments (1-1b) must execute in the correct procedural order to derive each **yj** value from the state-variable values and **t**, possibly using already computed **yi** values. An out-of-order assignment would incorrectly try to use defined-variable values from an earlier derivative call. The state equations (1-1a) are normally programmed following the defined-variable assignments (1-1b).

Conventional simulation programs such as ACSL[®] automatically sort the defined-variable assignments so that they use only **yi** values already computed during the current derivative call. If that is impossible due to an “algebraic loop,” the program returns an error message (sort error). DESIRE’s more general program system does not sort statements automatically. But in programs without subscripted variables or vectors (Chapter 3), DESIRE prevents assignments to undefined variables, and thus algebraic loops, by returning an error message (see also Section 3-5).³

EXAMPLES OF SIMPLE APPLICATIONS

1-10. Oscillators and Computer Displays

(a) A Linear Harmonic Oscillator

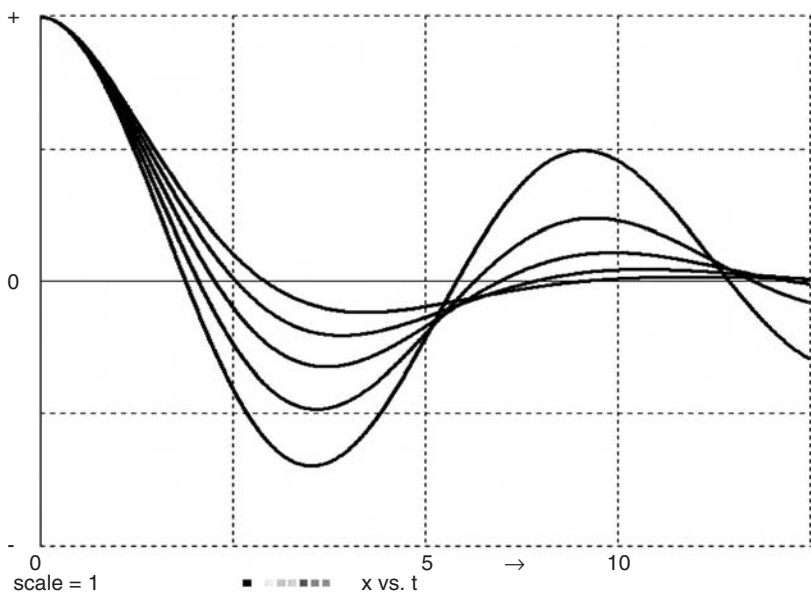
The complete small program in Figure 1-3 illustrates the main features of a DESIRE simulation. The DYNAMIC program segment following the **DYNAMIC** statement in Figure 1-3a defines our model. We have modeled a simple damped harmonic oscillator or mass–spring–dashpot system with the differential equations

$$\frac{d}{dt} x = \dot{x} \quad | \quad \frac{d}{dt} \dot{x} = -ww * x - r * \dot{x}$$

³ Recursive assignments that input the assigned-to variable on the right-hand side, as in

$$y = y + x \quad y = a * x1 + y * x2$$

do not ordinarily appear in differential-equation code. If they do, DESIRE does not flag them with error messages but considers them as difference equations and assigns **y** the initial value 0 (Sections 2-1 and 2-2).



A LINEAR OSCILLATOR

```
-----
display N1      |      display C8      |      display Q
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. This complete simulation program for a linear oscillator produces five simulation runs with different values of the damping coefficient *r*.

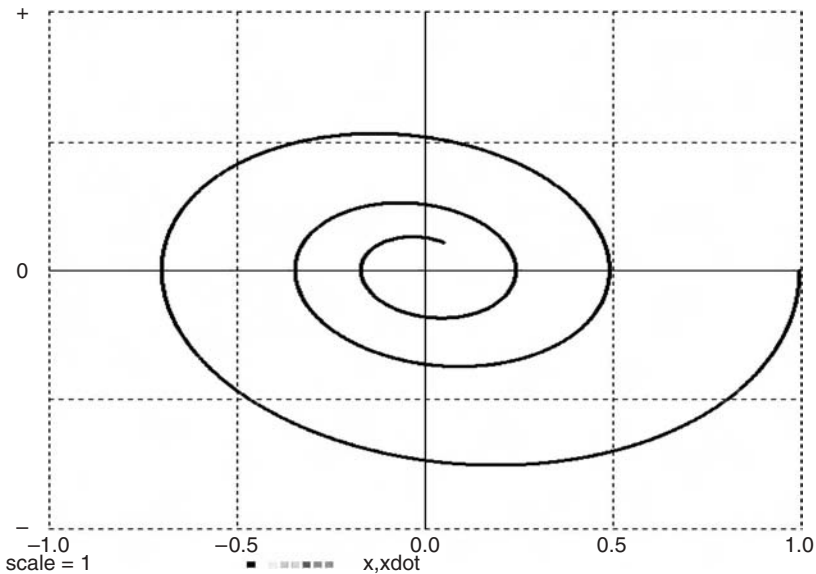


FIGURE 1-3b. A phase-plane plot (\dot{x} versus x) for the linear oscillator in Figure 1-3a.

We can add a display specification:

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

Model and display are exercised by the experiment-protocol script preceding the **DYNAMIC** statement. Successive experiment-protocol lines specify

- display colors and curve thickness
- 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

Initial values of time t and of the state variable \dot{x} are not specified and default to 0. The integration routine defaults to a fixed-step second-order Runge–Kutta rule.⁴

A simple experiment-protocol loop next calls for five simulation runs with five different values of the oscillator damping parameter r . The resulting displays are reproduced at the top of Figure 1-3a. Figure 1-3b shows a phase-plane plot.

⁴ The OPEN DESIRE reference manual in the book CD describes the complete program syntax, default values of different simulation parameters, and operating instructions.

(b) A Nonlinear Oscillator and Duffing's Differential Equation

The differential equations

$$d/dt \, x = xdot \quad | \quad d/dt \, xdot = -x * x * x - a * xdot$$

model an oscillator with a nonlinear spring. Figures 1-4a and b show the resulting time histories and a phase-plane plot obtained with $a = 0.02$. These results are clearly different from the linear-oscillator response in Figure 1-3. If we drive the nonlinear oscillator with a sinusoidal voltage $b \cos(t)$, we obtain Duffing's differential-equation system

$$d/dt \, x = xdot \quad | \quad d/dt \, xdot = -x * x * x - a * xdot + b * \cos(t)$$

Figure 1-4b shows solution displays and a program. The experiment-protocol script is interesting in that it first calls a simulation run to exhibit the initial transient, then a long simulation run with the display turned off to establish steady-state conditions, and finally a third run to display the steady-state solution.

Reference [9] discusses DESIRE programs for several other small physics problems.

1-11. Space Vehicle Orbits—Variable-step Integration

The space-vehicle orbit simulation in Figure 1-5 assumes a fixed earth exerting a simple inverse-square-law gravitational force on the satellite; effects of planets, moons, and so on are neglected. With the sun at the coordinate origin, the inverse-square-law accelerations in the x and y directions are

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

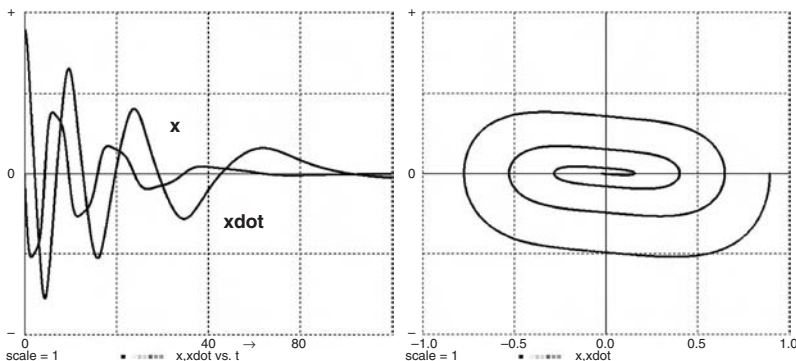
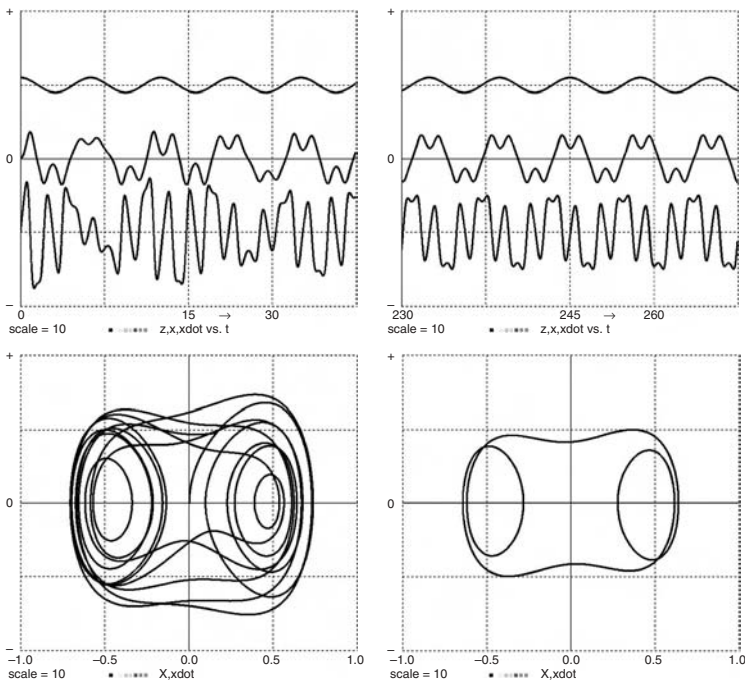


FIGURE 1-4a. Time histories and phase-plane plot for the nonlinear oscillator modeled with $d/dt \, x = xdot \quad | \quad d/dt \, xdot = -x * x * x - a * xdot + b * \cos(t)$.



--

DUFFING'S DIFFERENTIAL EQUATION

```

scale = 10 | display N1 | display C8 | display Q
a = 0.099 | b = 15
TMAX = 30 | DT = 0.0002 | NN = 10000
X = 0.02
drun
write "type go to continue" | STOP
TMAX = 200 | display 0 | drun
write " note how solution becomes periodic!"
TMAX = 30 | display 1 | drun

```

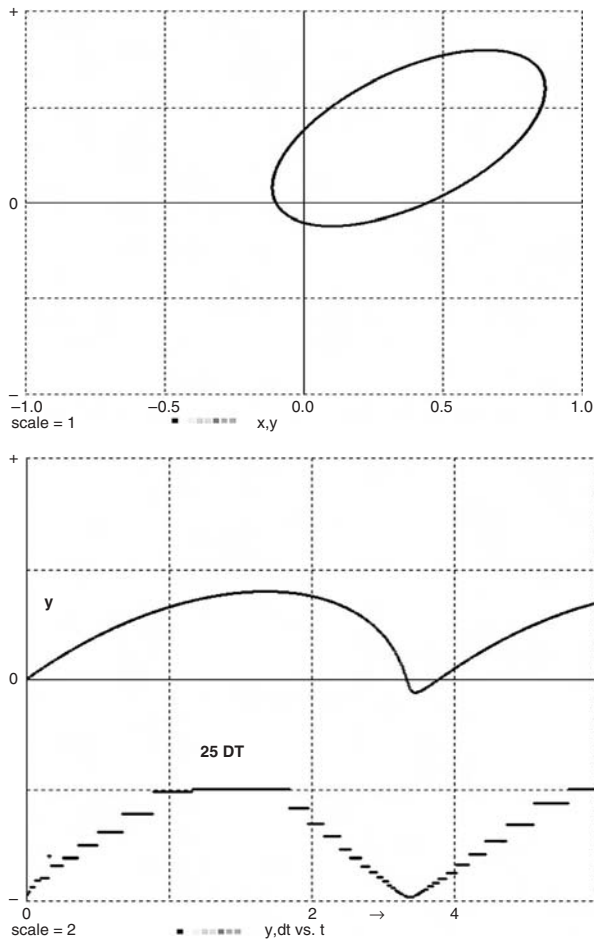
DYNAMIC

```

d/dt x = xdot | d/dt xdot = - a * xdot - x * x * x + b * cos(t)
--
z = cos(t)
Z = 0.5 * (z + scale) | X = 0.5 * x | XDOT = 0.5 * (xdot - scale)
dispt Z, X, XDOT

```

FIGURE 1-4b. A 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**), 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.



--

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.

With the program scaled so that the gravitational constant $\mathbf{a} = 1$, we obtain the simple differential-equation system⁵

$$\begin{array}{l|l} \mathbf{rr} = (\mathbf{x}^2 + \mathbf{y}^2)^{-1.5} & \\ \mathbf{d/dt\ x} = \mathbf{xdot} & \mathbf{d/dt\ y} = \mathbf{ydot} \\ \mathbf{d/dt\ xdot} = -\mathbf{x * rr} & \mathbf{d/dt\ ydot} = -\mathbf{y * rr} \end{array}$$

In Figure 1-5, an orbit starts around the sun to gain velocity for a trip to the outer planets. This involves dramatic velocity changes, and the small integration steps required during the high-velocity portion of the trajectory would slow the rest of the simulation. For this reason, such simulations employ an implicit variable-step/variable-order integration rule (**irule 15**). The second display in Figure 1-5 illustrates the integration-step changes.

1-12. A 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 (Section 7-1).

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

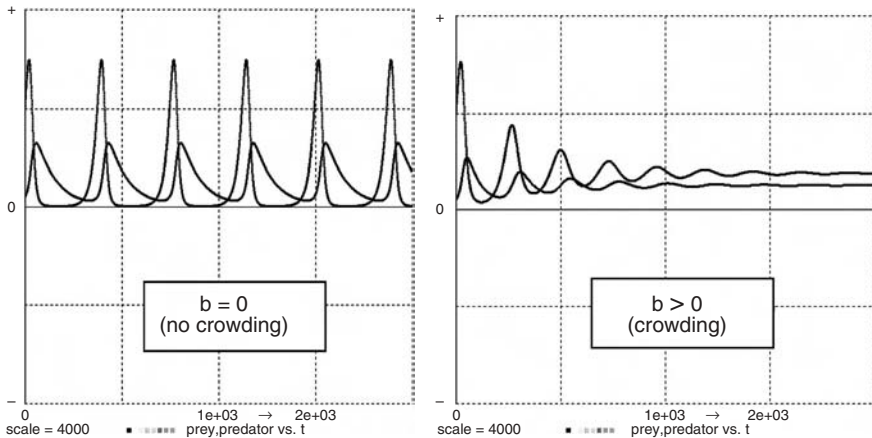
$$\begin{array}{l} \mathbf{d/dt\ prey} = (\mathbf{a1} - \mathbf{a4 * predator}) * \mathbf{prey} \\ \mathbf{d/dt\ predator} = (-\mathbf{a2} + \mathbf{a3 * prey}) * \mathbf{predator} \end{array}$$

The rate of increase of each population is proportional to the population size. $\mathbf{a1}$ 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 $\mathbf{a4 * predator}$ proportional to the size of the predator population (say a population of foxes). The predator population has a death rate $\mathbf{a2}$, and its birth rate $\mathbf{a3 * prey}$ is proportional to the prey population, which is its food supply.

⁵ This Cartesian-coordinate formulation is simpler than the polar-coordinate differential-equation system:

$$\begin{array}{l|l} \mathbf{x} = \mathbf{r * cos(theta)} & \mathbf{y} = \mathbf{r * sin(theta)} \\ \mathbf{d/dt\ r} = \mathbf{rdot} & \mathbf{d/dt\ rdot} = -\mathbf{GK/(r^2)} + \mathbf{r * thdot^2} \\ \mathbf{d/dt\ theta} = \mathbf{thdot} & \mathbf{d/dt\ thdot} = \mathbf{2 * rdot * thdot/r} \end{array}$$

The simulation program in Figure 1-6 demonstrates how easily such simple population-dynamics models can be modified. We added an extra predator death rate $\mathbf{b} * \text{predator}$ to account for the effect of crowding as the predator population increases and some predators kill one another. For $\mathbf{b} = 0$ (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 food supply. The



```
-- A PREDATOR-PREY PROBLEM
-- showing the effect of crowding
```

```
-----
display N1      | display C8
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. A population-dynamics simulation. For $\mathbf{b} = 0$ the program implements the classical Volterra–Lotka differential equations, which produce steady-state periodic fluctuations of the predator and prey populations. Positive values of \mathbf{b} model an increased predator death rate due to crowding, for example, by predator cannibalism. Predator and prey populations then converge to constant steady-state values.

number of rabbits then increases again, and the process repeats. But crowding ($\mathbf{b} > \mathbf{0}$) 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 Figure 1-7 models a billiard ball as a point (\mathbf{x}, \mathbf{y}) on a table bounded by elastic barriers at $\mathbf{x} = \mathbf{a}$, $\mathbf{x} = -\mathbf{a}$, $\mathbf{y} = \mathbf{b}$, and $\mathbf{y} = -\mathbf{b}$. For \mathbf{x}, \mathbf{y} within the barriers, the only acceleration is due to constant friction in the negative velocity direction, so that we program

$$\begin{aligned} d/dt \mathbf{x} &= \mathbf{x}dot & | & & d/dt \mathbf{y} &= \mathbf{y}dot \\ d/dt \mathbf{x}dot &= -\mathbf{fric} * \mathbf{x}dot/v & | & & d/dt \mathbf{y}dot &= -\mathbf{fric} * \mathbf{y}dot/v \end{aligned}$$

where the velocity \mathbf{v} is obtained with the defined-variable assignment

$$\mathbf{v} = \text{sqrt}(\mathbf{x}dot^2 + \mathbf{y}dot^2)$$

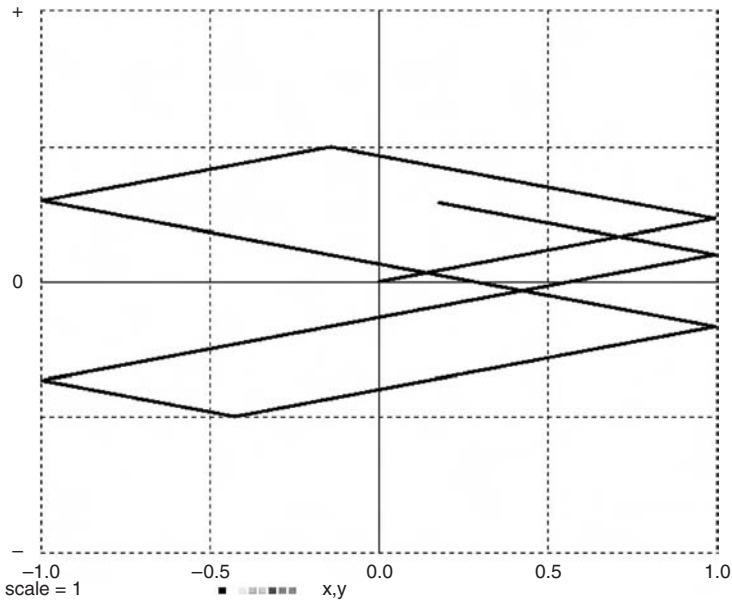
A differential-equation model of barrier impacts would need to formulate elastic and dissipative forces produced as the ball penetrates each barrier. This is not only complicated but involves very large accelerations and thus small integration steps. We neatly avoid these problems by terminating the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

$$\text{term } \text{abs}(\mathbf{x}) - \mathbf{a} \quad | \quad \text{term } \text{abs}(\mathbf{y}) - \mathbf{b}$$

The DESIRE experiment-protocol script then starts a new simulation run with the current position coordinates \mathbf{x}, \mathbf{y} and “reflected” velocity components $\mathbf{x}dot, \mathbf{y}dot$:

$$\begin{aligned} \text{if } \text{abs}(\mathbf{x}) > \mathbf{a} & \text{ then } \mathbf{x}dot = -\mathbf{R} * \mathbf{x}dot & | & & \mathbf{y}dot = \mathbf{R} * \mathbf{y}dot \\ & \text{else proceed} \\ \text{if } \text{abs}(\mathbf{y}) > \mathbf{b} & \text{ then } \mathbf{x}dot = \mathbf{R} * \mathbf{x}dot & | & & \mathbf{y}dot = -\mathbf{R} * \mathbf{y}dot \\ & \text{else proceed} \end{aligned}$$

where the restitution parameter \mathbf{R} measures the energy absorbed by the impact. A repeat loop continues this process until $\mathbf{t} > \mathbf{Tstop}$. The detailed syntax of **if/then/else** and **repeat/until** statements in DESIRE experiment-protocol scripts is found in the reference manual in the book CD. Figure 1-7 shows typical results as friction eventually brings the billiard ball to rest. The **display 2** statement keeps the program from erasing the display between runs.



--

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 $x = a$, $x = -a$, $y = b$, $y = -b$.

Similar run-splicing experiment-protocol scripts are useful in many other applications with radical switching operations, including simulations of electronic switching circuits. Reference [9] discusses several examples, including the classical bouncing-ball simulation and EUROSIM's peg-and-pendulum and switched-amplifier benchmarks.⁶

CONTROL-SYSTEM EXAMPLES

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

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

Figure 1-8 shows a simulation program. The sinusoidal servo input $\mathbf{u} = \mathbf{A} * \cos(\mathbf{w} * \mathbf{t})$ reduces to a step input for $\mathbf{w} = \mathbf{0}$. We model a simple linear controller with

$$\mathbf{voltage} = -\mathbf{k} * \mathbf{error} - \mathbf{r} * \mathbf{xdot} \quad (1-5)$$

Some nonlinear controllers will be discussed in Chapter 7. The controller gain \mathbf{k} and damping coefficient \mathbf{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.

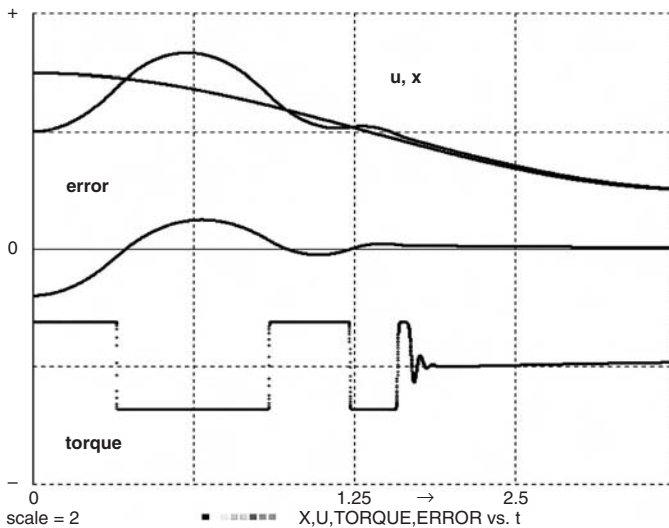
The motor voltage (1-5) produces a field current \mathbf{l} with a field-buildup delay modeled with

$$\mathbf{d/dt\ l} = -\mathbf{B} * \mathbf{l} + \mathbf{g1} * \mathbf{voltage} \quad (1-6)$$

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

$$\mathbf{torque} = \mathbf{maxtrq} * \tanh(\mathbf{g2} * \mathbf{l}/\mathbf{maxtrq}) \quad (1-7)$$

⁶ Reference [9] used an earlier version of DESIRE.



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 * V/maxtrq)
d/dt x = xdot | d/dt xdot = torque - R * xdot
```

```
--
```

scaled stripchart display

```
X = 5 * x + 0.5 * scale | U = 5 * (u + 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 $u = A \cos(w \cdot t)$. You can also set $w = 0$ to obtain the step response of the servomechanism.

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

$$\begin{aligned} d/dt \mathbf{x} &= \mathbf{x}\dot{\mathbf{d}}\mathbf{ot} \\ d/dt \mathbf{x}\dot{\mathbf{d}}\mathbf{ot} &= (\mathbf{torque} - \mathbf{R} * \mathbf{x}\dot{\mathbf{d}}\mathbf{ot})/\mathbf{M} \end{aligned} \quad (1-8)$$

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

The simulation program in Figure 1-8 sets system parameters and models the servomechanism with two defined-variable assignments (1-5) and (1-8) and three state differential equations (1-6) 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 use different test inputs $\mathbf{u(t)}$ similar to inputs for the intended application, for example, step inputs, ramps, sinusoids (or noise, as in Section 5-8). 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 $\mathbf{x(t)} - \mathbf{u(t)}$ for a given input $\mathbf{u(t)}$. One can, for instance, record the maximum of the absolute error or the squared error, as in Section 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,

$$\begin{aligned} d/dt \text{IAE} &= \text{abs}(\mathbf{x} - \mathbf{u}) && (\text{IAE, integral absolute error}) \\ d/dt \text{ISE} &= (\mathbf{x} - \mathbf{u})^2 && (\text{ISE, integral squared error}) \\ d/dt \text{ITAE} &= \mathbf{t} * \text{abs}(\mathbf{x} - \mathbf{u}) \\ d/dt \text{ISTAE} &= \mathbf{t}^2 * \text{abs}(\mathbf{x} - \mathbf{u}) \end{aligned}$$

where ISE/TMAX is the mean squared 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 system, for example, with a view to minimizing excessive space-vehicle accelerations. Parameter-influence studies are discussed in more detail in Sections 4-1 to 4-3.

1-15. Control-system Frequency Response

Simulation experiments can explore control-system frequency response with successive different sinusoidal inputs. For linear control systems, one

can instead simulate the system impulse response and program an experiment-protocol script to produce its Fourier transform [9]. DESIRE experiment-protocol scripts can perform fast Fourier transforms and work with complex numbers for frequency-response and root-locus plots [9]. The book CD shows a number of simple examples.

1-16. Simulation of a Simple Guided Missile

(a) A Guided Torpedo

Figure 1-9a shows a missile pursuing a target [19–22]. The problem is scaled so that **TMAX** = 1, and distances are in 1000-foot 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 torpedo 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 will be a guided torpedo. In water, drag and side forces are approximately proportional to the square **u**² of **u**. The accelerations along and perpendicular to the torpedo's longitudinal axis are then approximated by

$$\begin{aligned} (d/dt) u &= (\text{thrust} - \text{drag})/\text{mass} = UT - a_2 * u^2 \\ (d/dt) v &= b_1 * u^2 \sin \gamma + b_2 * u * \text{phidot} + b_3 * v * \text{rudder} \end{aligned}$$

The yaw-rotation equations are

$$\begin{aligned} (d/dt) \text{phi} &= \text{phidot} \\ (d/dt) \text{phidot} &= c_1 * u^2 * \sin \gamma + c_2 * u * \text{phidot} + c_3 * u^2 * \text{rudder} \end{aligned}$$

where **c1** and **c2** are hydrodynamic- 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 **g2** between longitudinal axis and velocity vector is so small that

$$\sin \gamma \approx \tan \gamma \approx v/u$$

and the equations of motion for our DYNAMIC program segment become

$$\begin{aligned} (d/dt) u &= UT - a_2 * u^2 \\ (d/dt) v &= u * (b_1 * v + b_2 * \text{phidot} + b_3 * \text{rudder}) \\ (d/dt) \text{phidot} &= u * (c_1 * v + c_2 * \text{phidot} + c_3 * \text{rudder}) \\ (d/dt) \text{phi} &= \text{phidot} \\ (d/dt) x &= u * \cos(\text{phi}) - v * \sin(\text{phi}) \\ (d/dt) y &= u * \sin(\text{phi}) + v * \cos(\text{phi}) \end{aligned}$$

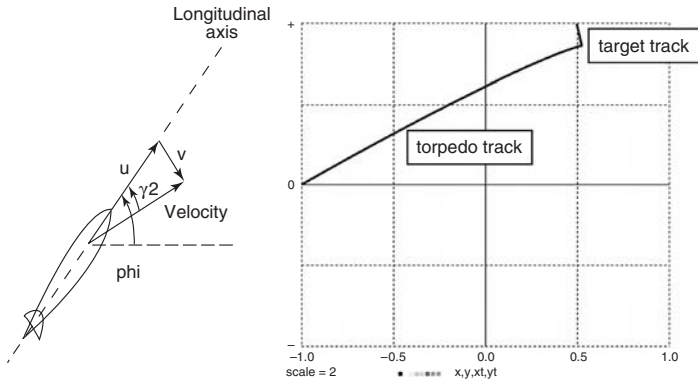


FIGURE 1-9a. A guided torpedo tracking a constant-speed target. The target angle **psi**, not shown here, is the angle between the horizontal line and the line joining the torpedo and target.

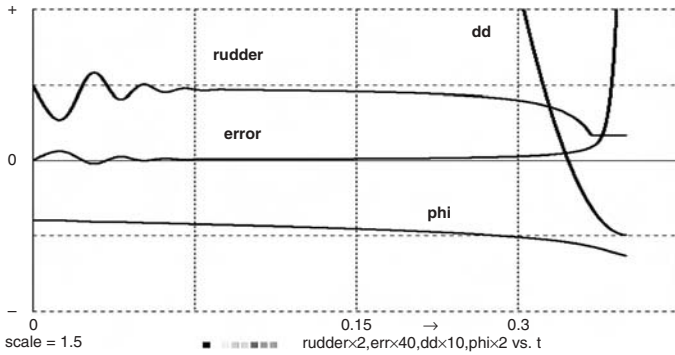


FIGURE 1-9b. Time histories of the torpedo rudder deflection, the error **phi-psi**, the angle **phi** and the squared distance **dd** to the target (see text).

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

$$\begin{aligned} x_t &= x_{t0} + v_{xt} * t & y_t &= y_{t0} + v_{yt} * t \\ \psi &= \arctan((y_t - y)/(x_t - x)) & dd &= (x - x_t)^2 + (y - y_t)^2 \end{aligned}$$

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 initially one is

```

--
--                                     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 | -- initial values
xt0 = 1 | yt0 = 2
rudder = 0
phi = atan2(yt0 - y, xt0 - x) | -- first aim at target
drunr
DYNAMIC
-----
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
-----
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)
--
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.

more or less directly behind the target (Fig. 1-10). More advanced guidance systems are discussed in Reference [18].

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

$$\text{error} = (\text{phi} - \text{psi}) \quad \text{rudder} = -\text{rumax} * \text{sat}(\text{gain} * \text{error})$$

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

$$\text{gain} = \text{gain0} + \text{A} * \text{t}$$

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) The Complete Simulation Program

Figure 1-9c lists the complete guided-torpedo program used to produce the displays in Figures 1-9a and b. The experiment protocol first selects an integration routine, display colors, and display scale, and 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 sets 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.

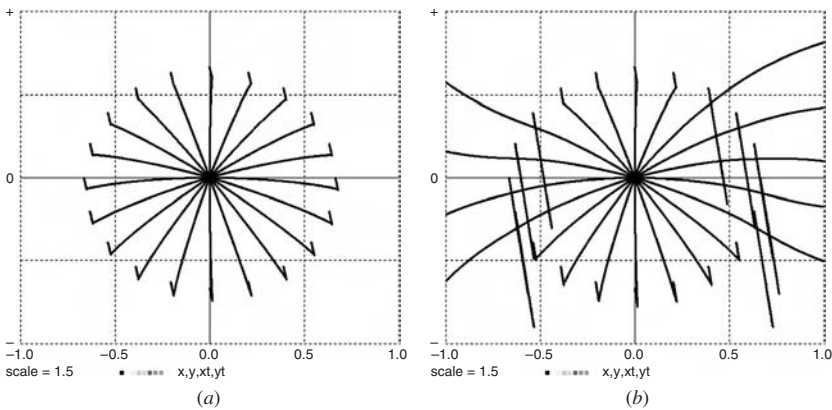


FIGURE 1-10. Multirun studies showing the results of torpedo shots at (a) low-speed and (b) high-speed targets appearing in different directions. It is a well-known fact [19,20] that the primitive pursuit-guidance scheme described in Section 1-16 can acquire a high-speed target only when the target track is ahead of the missile or behind it.

The DYNAMIC program segment following the **DYNAMIC** line begins with the defined-variable assignments. We specify the target coordinates **xt**, **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()** (Section 2-8a), which follows a **step** statement to ensure correct integration (Section 2-11).

Finally, the display command **DISPXY x, y, xt, yt** calls for simultaneous displays of the missile and target trajectories (**y** versus **x** and **yt** versus **xt**). Alternative display statements can plot time histories of **phi**, **psi**, **error**, and **rudder** (Fig. 1-9b). The simulation program can be loaded from a file or an editor window. Solution displays will then appear when a **run** command is typed.

WHAT DO WE DO WITH ALL THIS?

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

Simulations such as our torpedo example provide some 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: one begins to see how much more there is to know. Here are just a few questions that might come up:

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

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

evaluation of such results is an art rather than a science. It is the specific purpose of 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 are 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. U. M. Asher and L. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM Press, New York, 1998.
2. H. Elmquist, *Dymola User's Manual*, DynaSim A.B., Lund, Sweden, 2004.
3. L. Petzold, A Description of DASSL, a Differential-Algebraic-Equation Solver, in *Scientific Computing* (R. S. Stepleman, ed.), North Holland, Amsterdam, 1989.
4. J. Stoer, et al. *Introduction to Numerical Analysis*, Springer, New York, 2002
5. G. A. Korn and J. V. Wait, *Digital Continuous-System Simulation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
6. M. M. Tiller, *Introduction to Physical Modeling with Modelica*, Kluwer Academic Publishers (now Springer), New York, 2004.
7. P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley, New York, 2004.
8. *DYMOLA Manual*, Dynasim A.B., Lund, Sweden, 2005.
9. G. A. Korn, *Interactive Dynamic System Simulation with Microsoft Windows*, Taylor and Francis, London, 1998.
10. F. Cellier and E. Kofman, *Continuous-System Simulation*, Springer, New York, 2006.
11. C. W. Gear, DIFSUB, Algorithm 407, *Communications in ACM*, **14**, No. 3, 3/7, 1971.
12. G. K. Gupta et al., A review of recent developments in solving ODEs, *Computing Surveys*, **17**, March 1985, pp. 5–47.
13. E. Hairer et al., *Solving Ordinary Differential Equations* (2 vols.), Springer, Berlin, 1987.
14. A. C. Hindmarsh, LSODE and LSODI, *ACM/SIGNUM Newsletter*, **15**, No. 4, 1980.

15. J. D. Lambert, *Numerical Methods for Ordinary Differential Equations: The Initial-Value Problem*, Wiley, New York, 1991.
16. W. E. Schiesser, *A Comparative Study of Merson-type Runge-Kutta Algorithms*, Report, Chemical Engineering Department, Lehigh University, Bethlehem, PA, 1980.
17. L. F. Shampine and H. A. Watts, Software for Ordinary Differential Equations, in *Mathematical Software* (L. R. Crowell, ed.), Prentice-Hall, Englewood Cliffs, NJ, 1984.
18. J. C. Butcher, *The Numerical Analysis of Ordinary Differential Equations*, Wiley, Chichester, UK, 1980.
19. P. Garnell, *Guided Weapon Control Systems*, 2nd Ed., Brassey's Defence Publishers, London, 1980.
20. J. H. Blakelock, *Automatic Control of Aircraft and Missiles*, Wiley, New York, 1990.
21. R. G. Cottrell, Optimal intercept guidance, *AIAA J.* **9**, 1971, 1414–1415.
22. R. M. Howe, in *Hybrid Computation* (W. J. Karplus and G. A. Bekey, eds.), Wiley, New York, 1968.

2

Models with Difference Equations, Limiters, and Switches

SAMPLED-DATA ASSIGNMENTS AND DIFFERENCE EQUATIONS

2-1. Sampled-data Difference Equation Systems¹

Sampled-data assignments model applications such as digital filters, controllers, and neural networks. We recall that sampled-data assignments execute at the **NN** sampling points

$$t = t_0, t_0 + \text{COMINT}, t_0 + 2 \text{ COMINT}, \dots, t_0 + (\text{NN} - 1)\text{COMINT} = t_0 + \text{TMAX}$$

with

$$\text{COMINT} = \text{TMAX}/(\text{NN} - 1)$$

(Section 1-6). At each step, a sampled-data assignment input not already computed by a preceding assignment takes the value calculated at the last prior sampling point, starting with a given initial value. It follows that not all recursive sampled-data assignments are “algebraic loops” resulting from sort errors, as would be the case for continuous-variable assignments. Recursive

¹ Difference equations relating differential-equation-system variables (“continuous” or “analog” variables) will be treated in Section 2-16.

sampled-data assignments represent a difference-equation system whose solution is a set of successive output values generated by recursive substitution, starting with given initial values.

A differential-equation system such as Eq. (1-1) makes it obvious which variables are state variables and need initial values. This is not as easy in the case of difference-equation systems. We must draw on real knowledge of the model context to identify state variables and then execute sampled-data assignments into meaningful procedural order; otherwise they may mix past and present variable values into garbage.

Difference-equation state variables **q1, q2, ...** typically represent current and past values of significant model quantities **z1, z2, ...**; for instance,

$$\begin{aligned} \mathbf{q1} &= \mathbf{z1(t)}, \mathbf{q2} = \mathbf{z1(t - COMINT)}, \mathbf{q3} = \mathbf{z1(t - 2 COMINT)} \\ \mathbf{q4} &= \mathbf{z2(t)}, \mathbf{q5} = \mathbf{z2(t - COMINT)} \end{aligned}$$

A difference-equation system of order **N** relates current values (i.e., values at the time **t**) of **N** state variables to past values of these state variables (i.e., values at the time **t - COMINT**).

Just as in the case of a differential-equation system (1-1), we begin by computing various intermediate results and output quantities as functions of previously assigned state-variable values **qi**. Current values of such defined variables are produced by defined-variable assignments

$$\mathbf{pj} = \mathbf{Gj(t; q1, q2, \dots, qN; p1, p2, \dots)} \quad (\mathbf{j} = 1, 2, \dots) \quad (2-1a)$$

Note that defined-variable assignments relate current **pj** values on the left-hand side to past values of the state variables **qi** and to already computed current values of other defined variables **pj**. This means that the defined-variable assignments (2-1a) must be properly sorted into a procedural order that produces successive **pj** values without algebraic loops, just as in Section 1-9.²

Following the defined-variable assignments (2-1a), we compute the current values **Qi** of our state variables **qi** with **N** difference-equation assignments

$$\mathbf{Qi} = \mathbf{Fi(t; q1, q2, \dots, qN; p1, p2, \dots)} \quad (\mathbf{i} = 1, 2, \dots, \mathbf{N}) \quad (2-1b)$$

The functions on the right-hand side again use past values of the **qis** and current values of the **pj**.

² When there are no vector operations or subscripted variables, DESIRE again automatically prevents sort errors with “undefined variable” messages. When there are vector operations, we proceed as in Section 1-9.

Following the assignments (2-1a) and (2-1b), we must set new state-variable values **qi** for the next sampling time with **N** updating assignments

$$\mathbf{qi} = \mathbf{Qi} \quad (\mathbf{i} = 1, 2, \dots, \mathbf{N}) \quad (2-1c)$$

To summarize, a complete difference-equation program (2-1) starts with sorted defined-variable assignments (2-1a). These are followed by difference-equation assignments (2-1b) and then updating assignments (2-1c). We execute all these assignments, in that order, at successive sampling points. This solves the difference-equation system by recursive substitution of new **qi** values, starting with given initial values. Figure 2-1 shows an example (see also Fig. 2-4).

DESIRE normally does not provide default initial values for unsubscripted difference-equation state variables. Their initial values **qi = qi(t0 – COMINT)** must be explicitly assigned by the experiment protocol, even if they equal 0. Section 2-2 deals with an exception to this rule.

2-2. ‘Incremental’ Form of Simple Difference Equations

DESIRE automatically recognizes simple recursive assignments such as

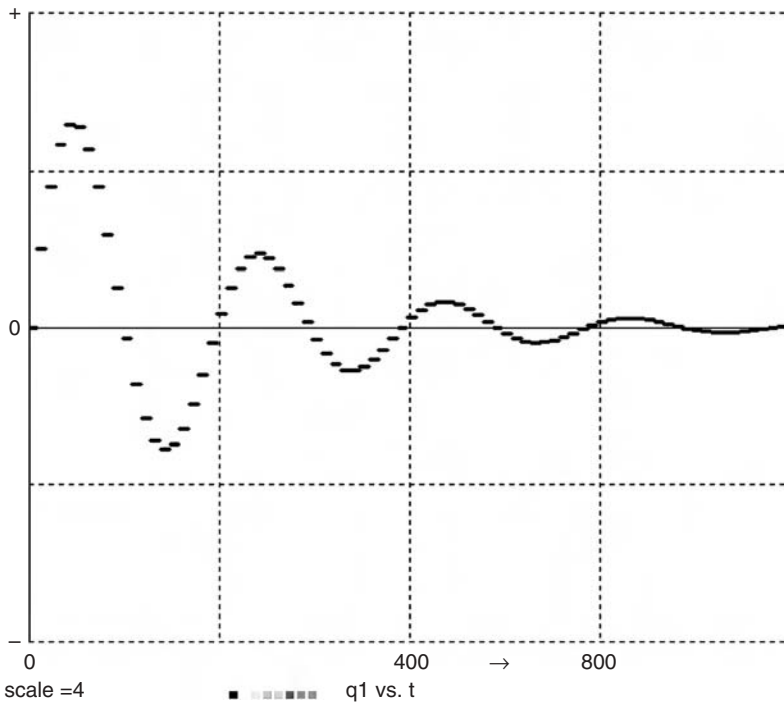
$$\mathbf{qi} = \mathbf{Fi}(\mathbf{t}; \mathbf{qi}) \quad (2-2)$$

as difference equations (see also Section 2-16). The program then treats **qi** as a difference-equation state variable, and automatically assigns **qi** the default initial value **qi(t0 – COMINT) = 0** (even though **t** starts at **t = t0**). **qi** is updated by recursive substitutions. In particular, **qi = qi + fi(t; qi)** produces **qi** as a sum of its initial value and successive increments **fi** (see also Section 4-6c). If the experiment protocol has assigned **qi** a nonzero initial value, say with **qi = 5**, this is correctly treated as **qi(t0 – COMINT)**.

The assignment (2-2) relates current **qi** values to past values without an updating assignment (2-1c). But when there is more than one difference equation, this scheme does not work if **Fi** depends, directly or indirectly, on a state variable other than **qi**. The solution then depends on the order in which the recursive assignments are evaluated and is likely to be garbage. This is precisely why we wrote each difference equation (2-1b) as an assignment to a “predictor variable” **Qi** rather than to **qi**. It is always safe to rewrite difference equations such as (2-2) as

$$\mathbf{Qi} = \mathbf{Fi}(\mathbf{t}; \mathbf{q1}, \mathbf{q2}, \dots; \mathbf{p1}, \mathbf{p2}, \dots)$$

and to program updating assignments **qi = Qi** following the difference equations as in Section 2-1.



NN = 801

a = 1.8 | b = - 0.90

t = 0 | q1 = 0 | q2 = 1

drun

DYNAMIC

SAMPLE 10 | -- for better display only

Q1 = q2 | Q2 = a*q2 + b * q1 | -- difference eqs.

q1 = Q1 | q2 = Q2 | -- update state vrbls.

FIGURE 2-1. Difference-equation program and display of q_1 versus t for a digital bandpass filter processing the time series $q(0), q(1), q(2), \dots$ ($t_0 = 0$, **COMINT** = 1). **SAMPLE 10** was used only to get a better display. Digital filters and controllers are important applications of difference equations.

2-3. Combining Differential Equations and Sampled-data Operations

As noted in Chapter 1, a DYNAMIC program segment can contain both differential-equation code and sampled-data code. Examples are simulations of digital controllers for analog plants (Sections 2-6 and 2-7) and differential-equation problems with pseudorandom noise inputs (Section 5-3). In such

programs, sampled-data assignments follow an **OUT** or **SAMPLE m** statement at the end of the differential-equation code, so that they execute only at periodic sampling times. As discussed in Section 1-8, properly designed integration routines admit sampling only at $t = t_0$ and at the end of integration steps.

Variables fed from a differential-equation system to a difference-equation system are defined variables. But all sampled-data inputs to differential-equation systems are state variables,³ for they relate past and present. In derivative calls between sampling points, these sampled-data inputs “hold” values assigned at the preceding sampling point. The experiment protocol must assign initial values to such sample-hold inputs. Otherwise, they will be flagged as undefined variables at $t = t_0$. In summary,

- Sampled-data assignments read inputs from the differential-equation section (simulated “continuous” or “analog” variables) computed at the current sampling time.
- In the “continuous” differential-equation section, the current value of each sampled-data input was produced at the preceding sampling time and stays constant until it is updated at the next sampling time. This models a sample-hold operation.

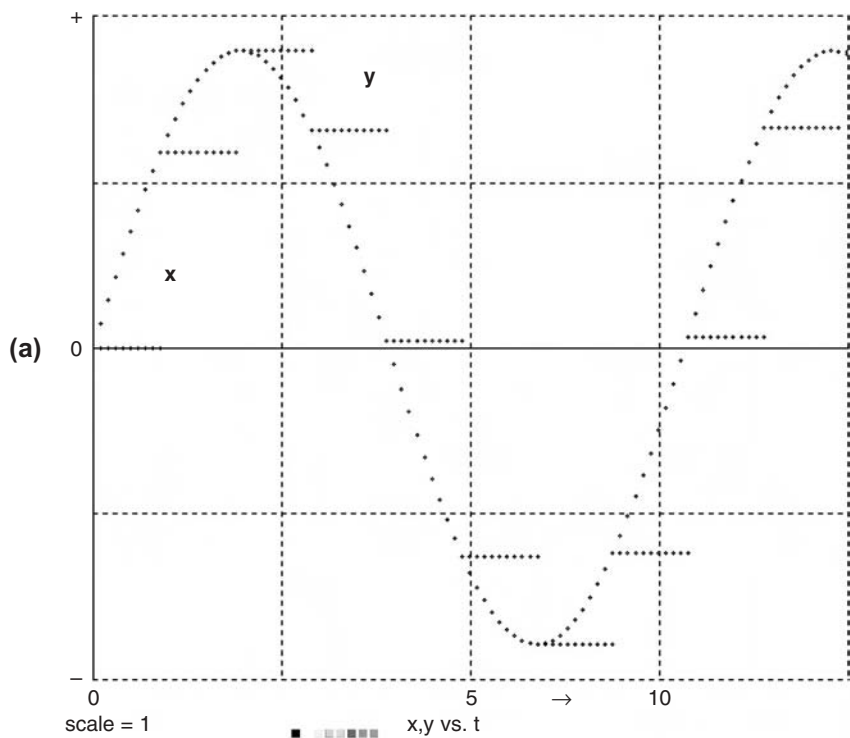
2-4. A Simple Example

Figure 2-2 illustrates the time relationships of data samples fed from a differential-equation system (“analog”) system to a sampled-data system (“digital” system) and back to the differential-equation system. Note that the analog input y equals the preceding sample of the sampled-data variable q . Figure 2-2a demonstrates the sample/hold action when $y = q$ is updated following a **SAMPLE m** statement with $m = 10$.⁴

³ Such “sample/hold inputs” to a differential equation system are state variables even if they are not difference-equation state variables.

⁴ It is not possible to see the sample/hold action for $m = 1$, for it would occur only between sampling points.

FIGURE 2-2. Data exchanges between a differential-equation (“analog”) system and a simple sampled-data (“digital”) system. There are no difference equations; q is a sample-hold state variable, not a difference-equation state variable. Graphic display (a) and output listing (b) were produced by the small DYNAMIC program segment in (c) for different values of **NN** and **m**. The experiment protocol has set $x = 0$ by default and explicitly assigned $q(0) = 0$. Note that the “analog” input y reads the q value from the preceding sampling step and is, therefore, always one step behind the current sampled-data result.



TMAX = 10 | NN = 101 | m = 10

	t	x	q	y
	0.00000e+000	0.00000e+000	0.00000e+000	0.00000e+000
	2.00000e+000	3.89418e-001	3.89418e-001	0.00000e+000
(b)	4.00000e+000	7.17356e-001	7.17356e-001	3.89418e-001
	6.00000e+000	9.32039e-001	9.32039e-001	7.17356e-001
	8.00000e+000	9.99574e-001	9.99574e-001	9.32039e-001
	1.00000e+001	9.09298e-001	9.09298e-001	9.99574e-001

TMAX = 10 | NN = 6 | m = 1

DYNAMIC

(c) $\frac{d}{dt} x = w * \dot{x}$ | $\frac{d}{dt} \dot{x} = -w * x$ | -- signal
 $y = q$ | -- D/A converter with sample/hold action,
-- holds the PRECEDING sample of q
SAMPLE m
 $q = x$ | -- A/D converter, reads CURRENT "analog" input x

2-5. Initializing and Resetting Sampled-data Variables

Unsubscripted sampled-data state variables and all sample-hold inputs to a differential equation system must be explicitly initialized by the experiment protocol to prevent “undefined variable” errors at $t = t_0$ (see Section 2-2 for special exceptions). Subscripted variables are necessarily defined by array declarations (Section 3-1) and default to 0.

Programmed and command-mode **reset** and **drunr** statements reset the system variables **t** and **DT** and all differential-equation state variables to their initial values at the start of the current simulation run. But **reset** and **drunr** do not reset difference-equation state variables or sampled-data inputs to differential-equation systems. These must be explicitly reset in the experiment-protocol script, perhaps with a named procedure collecting all such reset operations.

EXAMPLES OF MIXED CONTINUOUS/SAMPLED-DATA SYSTEMS

2-6. The Guided Torpedo with Digital Control

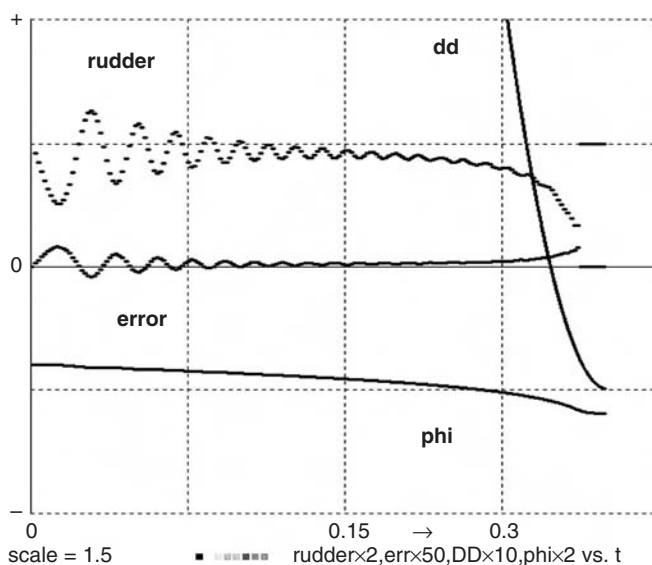
As a simple example, Figure 2-3 shows how the guided torpedo program of Section 1-16a can be modified to incorporate digital control. The controller operations

```
error = (phi - psi) * swtch(dd - DD)
gain = gain0 + 800 * t
rudder = - rumax * sat(gain * error)
```

become sampled-data assignments⁵ programmed following a **SAMPLE m** statement at the end of the **DYNAMIC** program segment. No difference equations are used. The first sampled-data assignment models analog-to-digital conversion of the continuous (analog) variables **phi** and **psi**. The other assignments represent controller operation and digital-to-analog conversion; **error** and **gain** are intermediate results. The simulated controller feeds its output **rudder** to the differential-equation system as a sample-hold input. **rudder** must be explicitly initialized at $t = 0$ (Section 2-3).

The controller sampling rate is $(NN - 1)/(m * TMAX)$. With sufficiently large sampling rates, the simulation results are similar to those in Section 1-16 (Figure 2-3).

⁵ Note that **swtch(dd - DD)** and **sat(gain * error)** appear in sampled-data assignments, so that they switch only at sampling times and cannot hurt numerical integration (Sections 2-9 and 2-10).



DYNAMIC

```

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
--
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)
--
term rr - dd
    
```

```

SAMPLE m |  --  digital controller
error = (phi - psi) * swtch(dd - DD)
gain = gain0 + 800 * t
rudder = - rumax * sat(gain * error)
    
```

FIGURE 2-3. Time-history display and DYNAMIC program segment for the digitally controlled torpedo (see also Fig. 1-9). Sampled-data operations are programmed following the **SAMPLE m** statement that sets the sampling rate. The sampled-data variable **rudder** must be initialized by the experiment protocol.

2-7. Simulation of a Plant with a Digital PID Controller

The simple digital controller in Section 2-6 involved no recursive sampled-data assignments, but we shall next study a true difference-equation controller. The program in Figure 2-4 models digital PID (proportional/integral/derivative) control [1] of an analog plant represented by differential equations similar to those for the servo in Section 1-14, that is,

$$\begin{aligned} \text{torque} &= \text{maxtrq} * \tanh(y/\text{maxtrq}) \\ d/dt \text{ c} &= \text{cdot} \quad d/dt \text{ cdot} = 10 * \text{torque} - R * \text{cdot} \end{aligned}$$

Torque saturation is again represented by the **tanh** function. The program neglects analog-to-digital converter quantization, but this could be implemented as shown in Section 2-15.

The simulated digital controller samples the analog input **u** and the analog output **c** (this models analog-to-digital conversion) to produce the sampled-data variable **error**. For simplicity, we specified a constant input **u = 0.7**. The controller then computes the sampled-data error measure **error = c - u**. To produce the controller output

$$y = B0 * q1 + B1 * q2 + B2 * (q2 - \text{error})$$

we must solve the difference-equation system

$$\begin{aligned} Q1 &= q2 \quad Q2 = q2 - \text{error} \\ q1 &= Q1 \quad q2 = Q2 \end{aligned}$$

for the state variables **q1**, **q2**.⁶ **y**, **q1**, and **q2** must be initialized by the experiment protocol.

An implied digital-to-analog converter converts **y** to an analog voltage that controls the motor torque **torque**. In the DYNAMIC program segment of Figure 2-4, the simulated digital controller (lines following the **SAMPLE m** statement) updates the state difference equations at every **mth** communication point, exactly as a real digital controller would. The sample rate is

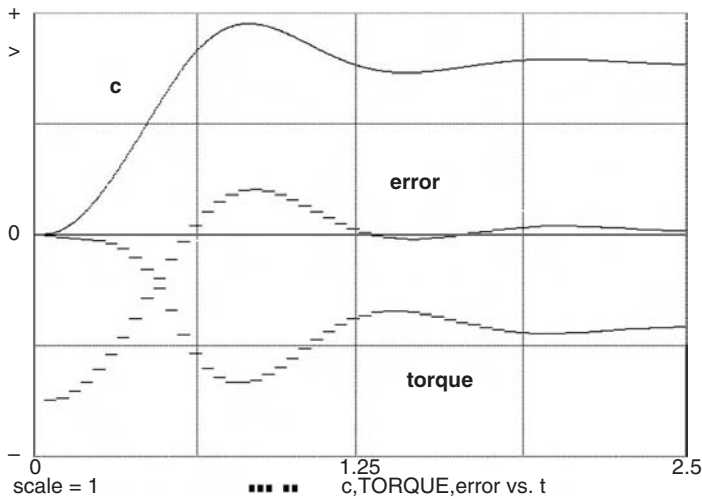
$$(NN - 1)/(m * TMAX) = 1/TS.$$

⁶ Reference 1 shows that the digital PID-controller has the z-transfer function [1]

$$G(z) \equiv KP + \frac{1}{2} (KI + TS) \frac{z + 1}{z - 1} + \frac{KD(z-1)}{TSz} \equiv \frac{Az^2 + Bz + C}{z(z-1)}$$

where **KP**, **KI**, and **KD** are the proportional, derivative, and integral gain parameters. Our program saves computing time by precomputing the PID parameters

$$B0 = KD/TS, \quad B1 = -KP + 0.5 * KI * TS - 2 * B, \quad B2 = KP + 0.5 * KI * TS + B0$$



-- "ANALOG" PLANT WITH A DIGITAL PID CONTROLLER

TMAX = 2.5 | DT = 0.001 | NN = 700 | display N1 | display C8

TS = 0.05 | -- the simulated sampling rate is 1/TS
m = TS * (NN - 1)/TMAX | -- display points per sample

u = 0.7 | -- step input
maxtrq = 0.8 | R = 3 | -- motor parameters

-- initial t, c, cdot all default to 0

x1 = 0 | x2 = 0

y = 0 | -- must initialize y

-- precompute P.I.D. parameters

KP = 3 | KI = 1.2 | KD = 0.2

B0 = KD/TS | B1 = - KP + 0.5 * KI * TS - 2 * KD/TS

B2 = KP + 0.5 * KI * TS + KD/TS

--

drun

DYNAMIC

torque = maxtrq * tanh(y/maxtrq) | -- analog plant

d/dt c = cdot | d/dt cdot = 10 * torque - R * cdot

SAMPLE m | -- digital controller

error = c - u

y = B0 * q1 + B1 * q2 + B2 * (q2 - error) | -- controller output

Q1 = q2 | Q2 = q2 - error | -- difference equations

q1 = Q1 | q2 = Q2 | -- update state variables

FIGURE 2-4. Simulation of an “analog” plant with a digital controller. Display commands are not shown.

MODELING LIMITERS AND SWITCHES

2-8. Limiters, Switches, and Comparators

The piecewise-linear library functions listed in Figure 2-5 work both in experiment-protocol scripts and DYNAMIC program segments. These functions are used in many engineering applications (see also [2–4]).

(a) Limiter Functions

lim(x) is a simple unit-gain limiter or half-wave rectifier (see also Section 2-13). The unit-gain saturation limiter **sat(x)** limits its output between -1 and 1 , and **SAT(x)** limits its output between 0 and 1 . More general unit-gain saturation limiters are obtained with

$$y = a * \text{sat}(x/a) \quad (\text{limits between } -a \text{ and } a > 0) \quad (2-3)$$

$$y = \text{lim}(x - \text{min}) - \text{lim}(x - \text{max}) \quad (\text{limits between } \text{min} \text{ and } \text{max} > \text{min}) \quad (2-4)$$

It is possible to approximate any reasonable continuous function of x as a sum of simple limiter functions,

$$a_0 + a_1 * \text{lim}(x - x_1) + a_2 * \text{lim}(x - x_2) + \dots \quad (2-5)$$

(b) Switching Functions and Comparators

The switch function **swtch(x - a)** in Figure 2-5b switches between 0 and 1 when $x = a$ (see also Section 2-16). Combining two **swtch** functions,

$$u = \text{swtch}(t - t_1) - \text{swtch}(t - t_2) \quad (t_1 < t_2) \quad (2-6)$$

we obtain a unit-amplitude pulse **u(t)** starting at $t = t_1$ and ending at $t = t_2$. $y = v * u$ models the result of switching the function **v(t)** *on* at $t = t_1$ and *off* at $t = t_2$.

Referring again to Figure 2-5b, **swtch(x)** and **sgn(x)** model the transfer characteristics of comparators that switch their output when their input x crosses zero. The useful function

$$y = \text{minus} + (\text{plus} - \text{minus}) * \text{swtch}(x - a) \quad (2-7)$$

models a relay comparator or function switch. Its output y switches between the values **minus** and **plus** when the input variable x crosses the comparison

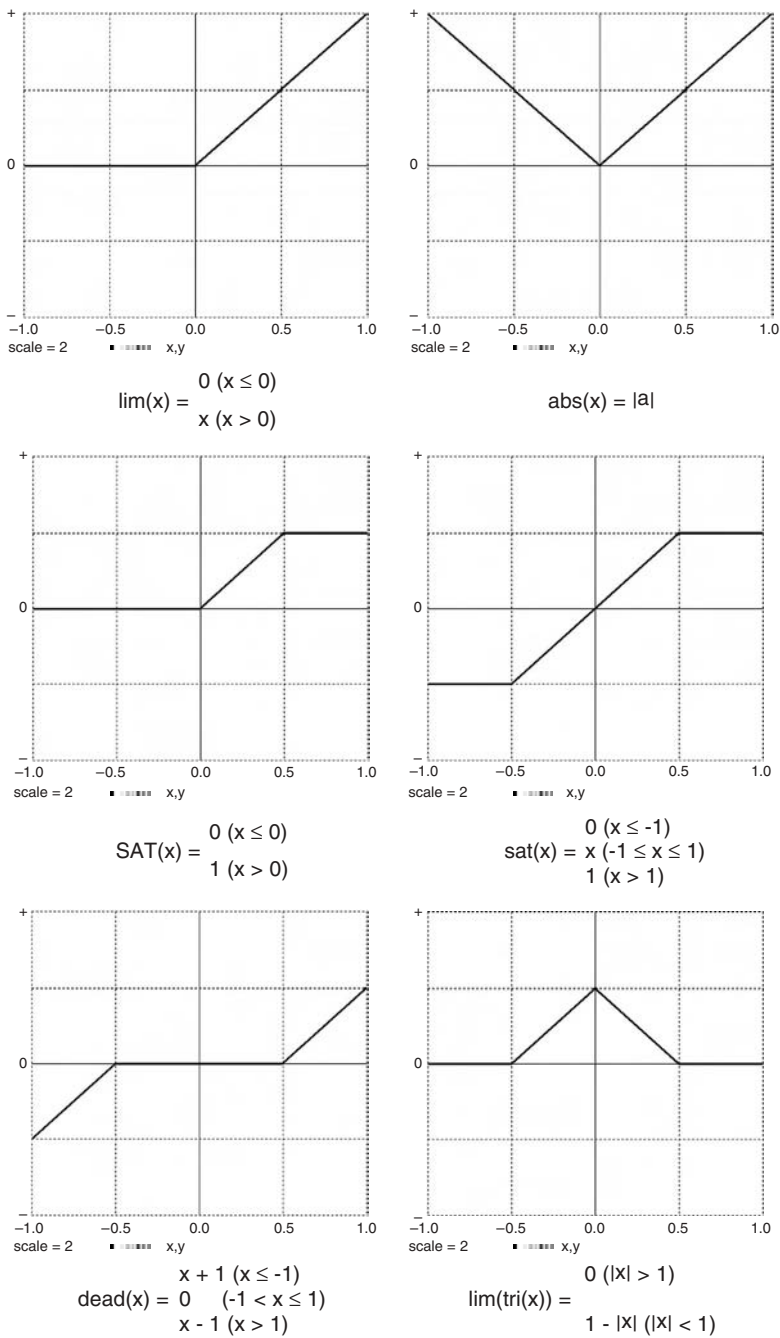


FIGURE 2-5a. Limiter functions.

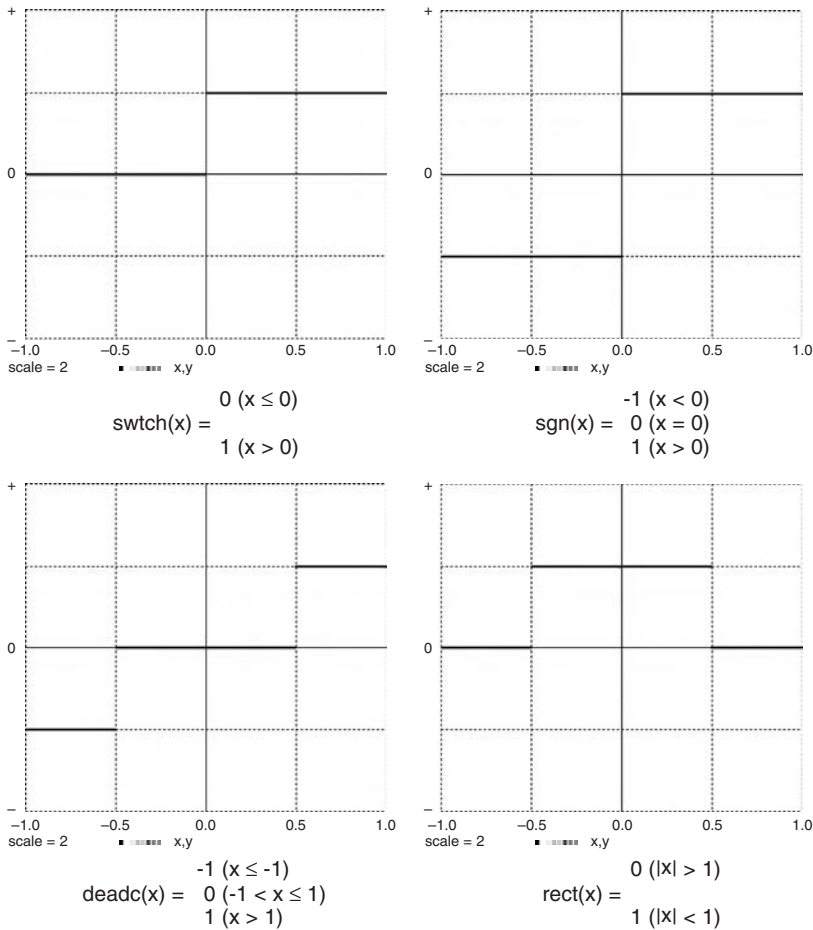


FIGURE 2-5b. Switching functions.

level **a**, **a**, **minus**, and **plus** can be variable expressions. A relay comparator can also be modeled with the library function

$$\text{comp}(x, \text{minus}, \text{plus}) = \begin{cases} \text{plus} & (x > 0) \\ \text{minus} & (x \leq 0) \end{cases}$$

The library function **deadc(x)** represents a comparator with a deadspace between $x = -1$ and $x = 1$. The function

$$y = \text{minus} * \text{switch}(a - x - \text{delta}) + \text{plus} * \text{switch}(x - a - \text{delta}) \quad (2-8)$$

implements a relay comparator with the symmetrical deadspace $\pm \text{delta}$.

2-9. Numerical Integration of Switch and Limiter Outputs, Event Prediction, and Display Problems

Switch-function outputs are discontinuous step functions, and limiter outputs have discontinuous derivatives.⁷ Numerical-integration steps must not cross such discontinuities, which violate the smooth-interpolation assumptions underlying all higher-order integration formulas. We already encountered the same problem with sampled-data integrands and solved it by providing integration routines that never step across the periodic sampling points (Section 1-8). But switch and limiter functions used in differential-equation problems will not, in general, switch at known periodic sampling times. To ensure correct numerical integration we must, therefore, modify either integration steps or switching times.

Early simulation projects simply reduced the integration step size **DT**, typically with a variable-step Runge–Kutta routine, and then ignored the problem. This often works (perhaps because models of stable control systems tend to reduce computing errors), but it is not the way to get reliable results. In particular, variable-step integration may fail at the switching points as it tries to decrease the integration-step size. Models requiring frequent switching (e.g., models of solid-state ac motor controllers) are especially vulnerable [5,6]. The situation is worse when simulations of mechanical and electrical systems involve several switching devices.

Two alternative methods can produce correct integration:

1. Some simulation programs predict the time **Tevent** when a function such as **switch(x)** will switch by extrapolating future values of **x** from a number of past values. The integration routine is then designed to force the nearest integration step to end at **t = Tevent**. The software must select the first function likely to switch, and the extrapolation formula must be as accurate as the integration rule [6–9].
2. We can execute program lines containing switch and limiter functions only at the end of integration steps (Section 2-11). This involves a compromise between switching-time resolution and integration step size; small integration steps slow down the computation.

The following sections describe two simple schemes for correct integration, but another problem remains. Computer displays cannot correctly display switched functions that switch more than once between display

⁷ Sometimes one can replace switch or limiter functions with smooth approximations. One can, for instance, approximate **sat(x)** with **tanh(a * x)**. Note that this technique also requires small integration steps.

sampling points. The only way to avoid multiple switching between display points is to increase the number of display points **NN** (or **NN/MM**, Section 1-6) and thus the minimum number of integration steps (Section 1-8). This display problem, though, does not affect computing accuracy, and continuous functions will display correctly.

2-10. Using Sampled-data Assignments

Since DESIRE integration routines never step across the periodic sampling points (1-2), all is well when switch and limiter operations are sampled-data assignments following an **OUT** or **SAMPLE m** statement at the end of the DYNAMIC program segment (Section 1-6). That is true, for instance, in simulated digital controllers.

In principle, all switch and limiter operations can be modeled as sampled-data assignments with a sufficiently high sampling rate. To obtain a desired switching-time resolution, one is then likely to require a sampling rate different from the input/output sampling rate used, for example, for displays. One can easily implement slower sampling with **SAMPLE m** or faster sampling by setting the DESIRE system variable **MM** to values greater than 1 (Section 1-6). In the latter case, the number of output samples for displays or listings will be less than **NN**, so that one cannot observe the switch output itself, only its effects on slower model variables (see also Section 2-9).

This simple solution of the switching problem again implies a compromise between switching-time resolution and computing speed, for no integration step can be larger than the sampling interval **COMINT = TMAX/(NN - 1)** (Section 1-9). This may be wasteful when we need only a few switch and/or limiter operations.

2-11. Using the step Operator and Heuristic Integration-step Control

A better way to obtain correct integration of switch and limiter functions is to program all such operations following a DESIRE **step** statement placed at the end of the differential equation program section. Sampled-data assignments following **OUT** and/or **SAMPLE m**, if any, would then be programmed after **step** assignments.

Assignments following the DESIRE **step** statement do not execute at every derivative call but only at **t = t0** and at the end of every integration step. The experiment protocol must initialize the targets of assignments following **step**, for they would otherwise be undefined at **t = t0**. They are, in fact, state variables relating past and present, just like sampled-data inputs.

Use of the **step** statement clearly solves our problem. As we already noted in Section 2-9, proper switching-time resolution requires the experiment protocol to set a sufficiently low value of **DT** for fixed-step integration rules, or of **DTMAX** or **TMAX/(NN – 1)** for variable-step-integration rules.⁸

But we can do much better. DESIRE integration rules 2, 3, and 5 (respectively Euler and fourth- and second-order Runge–Kutta rules) permit user-programmed changes of the integration step **DT** during simulation runs. We can thus start with some desired value **DT = DT0** and reduce **DT** heuristically when we are close to a switching time, for example, when the absolute value of a servo error is small. This technique reduces the computing-time loss, especially for simulations that need only occasional switching or limiting.

2-12. Example: Simulation of a Bang-bang Servomechanism

The bang-bang servomechanism modeled in Figure 2-6a is identical with the continuous-control servo in Section 1-14, except that now the control voltage does not vary continuously but switches between positive and negative values. We programmed the assignment

voltage = – sgn(k * error + r * xdot – 0.01* voltage)

following a **step** statement at the end of the DYNAMIC segment. For added realism, we implemented a Schmitt trigger (Section 2-16e) instead of a simple comparator by subtracting a fraction of **voltage** in the **sgn** argument.

The experiment protocol script sets an initial value for **voltage**, which would otherwise be undefined at **t = 0**. DESIRE’s integration rule 5 (**irule 5**) implements second-order Runge–Kutta integration and allows one to program

DT = DT0 * SAT(abs(error * pp)) + DTMIN

where **DT0**, **DTMIN**, and **pp** are parameters set by the experiment protocol. **DT** decreases to **DTMIN** when the servo error **error** is small. Figure 2-6a lists the program, and Figure 2-6b shows results, including the interesting time history of the programmed integration step.

If there is more than one discontinuous function, two or more **DT** expressions must be multiplied together

⁸ DESIRE integration rules 4–8 let one set **DTMAX** explicitly. For integration rules 9–15, we resort to the method of Section 2-11 and make **NN** large enough to obtain the desired time resolution. We can use **MM > 1** to get more sampling points than input/output points.

```

--      BANG-BANG SERVOMECHANISM
--      ensures correct integration with step operator
--
--      DT is programmed heuristically with irule 5
-----
irule 5 | --      permits user-programmed DT
-----
scale = 2 | display N1 | display C8 | --      display
TMAX = 2.5 | NN = 10000
-----
A = 0.1 | w = 1.2 | --      signal parameters
B = 100 | maxtrq = 1 | --      motor parameters
g1 = 10000 | g2 = 1 | R = 0.6
k = 40 | r = 2.5 | --      control parameters
--
pp = 100 | DT0 = 0.0002 |      DTMIN = DT0/10
-----
voltage = 0 | --      must initialize this!
drun
write "maxDT = ";DT0 + DTMIN
-----
DYNAMIC
-----
u = A * cos(w * t) | --      input
error = x - u | --      servo error
torque = maxtrq * tanh(g2 * V/maxtrq)
-----
d/dt V = - B * V + g1 * voltage | --      motor field delay
d/dt x = xdot |
d/dt xdot = torque - R * xdot
-----
step
voltage = - sgn(k * error + r * xdot - 0.01 * voltage)
DT = DT0 * SAT(abs(error * pp)) + DTMIN
--
-----      rescaled stripchart display
--
X = 5 * x + 0.5 * scale | U = 5 * u + 0.5 * scale
ERROR = 4 * error
TORQUE = 0.25 * torque - 0.5 * scale
dt = 2500 * DT- scale
dispt X, U, TORQUE, ERROR, dt

```

FIGURE 2-6a. DESIRE program for the bang-bang servomechanism.

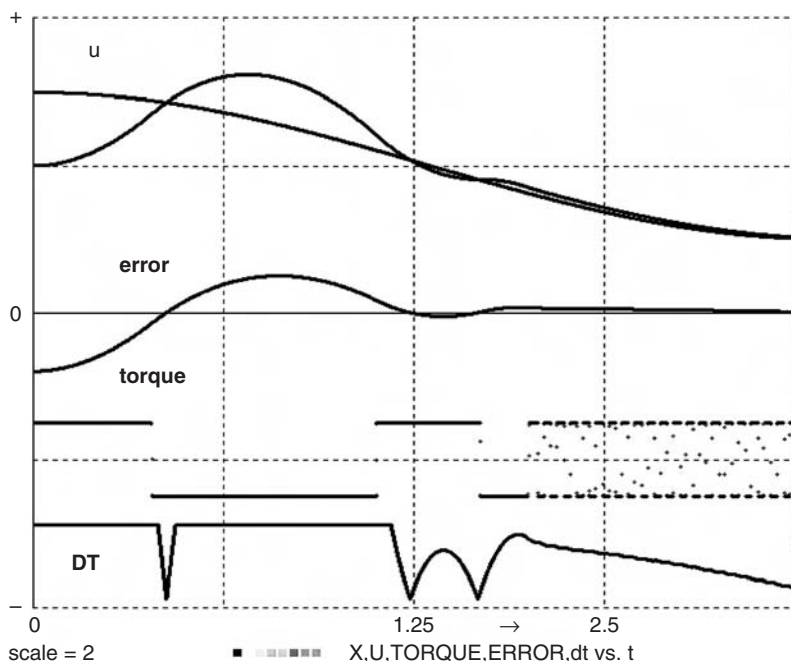


FIGURE 2-6b. Scaled stripchart display for the bang-bang servomechanism. The display shows time histories of the input u , output x , servo error, motor torque, and programmed time step DT . The original display showed each curve in a different color.

LIMITERS, SWITCHES, AND DIFFERENCE EQUATIONS

2-13. Limiters, Absolute Value, and Maximum/Minimum Selection

In most digital computers, the fastest nonlinear floating-point operation is not the simple limiter function (Section 2-8; see also [2-4,10]) but the absolute-value function

$$\text{abs}(x) \equiv |x| = \begin{cases} -x & (x < 0) \\ x & (x \geq 0) \end{cases} \quad (2-9)$$

(full-wave rectifier), which only needs to change the sign bit of a floating-point number. It is therefore profitable to remember the relations

$$\lim(x) \equiv 0.5 * [x + \text{abs}(x)] \equiv 0.5 * x + \text{abs}(0.5 * x) \quad (2-10)$$

$$\text{sat}(x) \equiv \lim(x + 1) - \lim(x - 1) - 1 \equiv 0.5 * [\text{abs}(x + 1) - \text{abs}(x - 1)] \quad (2-11)$$

$$\text{SAT}(x) \equiv \lim(x) - \lim(x - 1) \equiv 0.5 * [1 + \text{abs}(x) - \text{abs}(x - 1)] \quad (2-12)$$

$$\text{deadz}(x) \equiv x - \text{sat}(x) \equiv x - 0.5 * [\text{abs}(x + 1) - \text{abs}(x - 1)] \quad (2-13)$$

$$\text{tri}(x) \equiv 1 - \text{abs}(x) \quad \lim[\text{tri}(x)] \equiv \text{tri}[\text{sat}(x)] \equiv \text{TRI}(x) \quad (2-14)$$

These identities are, in fact, used to implement DESIRE's library functions. To find the largest and smallest of two arguments **x**, **y**, we use

$$\begin{aligned} \max(x, y) &\equiv x + \lim(y - x) \equiv y + \lim(x - y) \\ &\equiv 0.5 * [x + y + \text{abs}(x - y)] \end{aligned} \quad (2-15a)$$

$$\begin{aligned} \min(x, y) &\equiv x - \lim(x - y) \equiv y - \lim(y - x) \\ &\equiv 0.5 * [x + y - \text{abs}(x - y)] \end{aligned} \quad (2-15b)$$

Note also

$$\max(x, y) - \min(x, y) \equiv x + y \quad (2-16)$$

$$\lim(x) \equiv \max(x, 0) \quad (2-17)$$

2-14. Output-limited Integration

Integration of the switched function

$$\begin{aligned} \text{ydot} &= \text{swtch}(\max - y) * \lim(x) \\ &+ \text{swtch}(y - \min) * \lim(-x) \quad (\min < \max) \end{aligned} \quad (2-18)$$

stops whenever the integral **y** produced by **d/dt y = ydot** exceeds preset bounds [8]. Note that this is not the same as an integrator followed by an output limiter.

2-15. Modeling Signal Quantization

The model digital controllers in Sections 2-6 and 2-7 processed ordinary floating-point numbers [4]. But one may want to study the effects of signal quantization in digital control systems or in simulated signal processors and

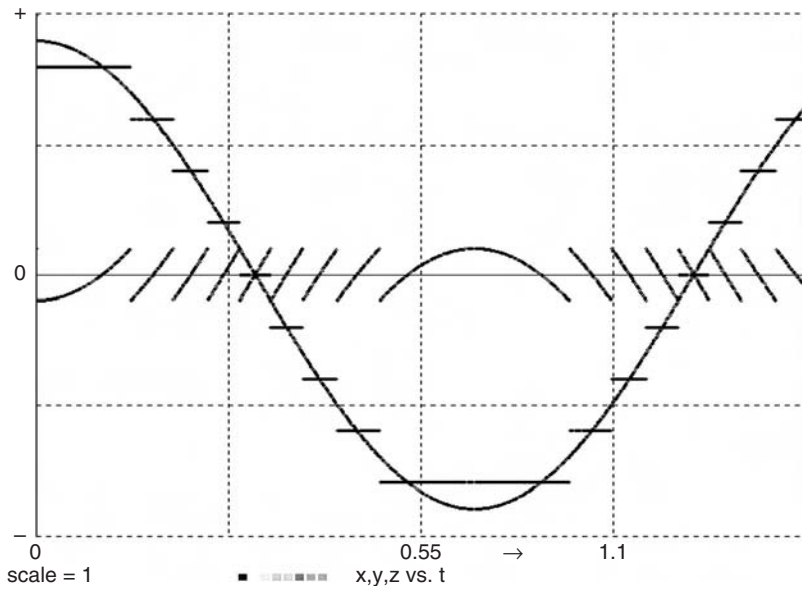


FIGURE 2-7. Signal quantization and quantization noise.

digital measurement systems. Figure 2-7 illustrates quantization of a sine wave with the assignment

$$y = a * \text{round}(x/a)$$

where **a** is the quantization interval. The error $y - x$ caused by signal quantization is the quantization noise.[4] The DESIRE library function **round(x)** returns floating-point numbers rounded to the nearest integer value, not integers. **round(x)** is a switched step function that needs to follow a **step**, **OUT**, or **SAMPLE m** at the end of a DYNAMIC program segment. **round(x)** can also implement rounding in experiment-protocol scripts.

2-16. Continuous-variable Difference Equations with Switching and Limiter Operations

(a) Introduction

This section presents a number of powerful modeling tricks that implement simple recursive assignments

$$q = F(t; q) \quad (2-19)$$

in DYNAMIC program segments. We already discussed sampled-data assignments of this form in Section 2-2. But **q** is not necessarily a sampled-data

variable; \mathbf{q} can be a “continuous” variable used in a differential-equation system. In any case, DESIRE recognizes recursive assignments (2-19) as difference equations and automatically assigns the difference-equation state variable \mathbf{q} the default initial value 0, as in Section 2-2. As already noted for sampled-data state variables (Section 2-5), difference-equation state variables are not automatically reset by **reset** or **drunr** statements. The experiment protocol must reset them explicitly as needed.

If the function \mathbf{F} in Eq. (2-19) involves limiters or switches (as in the following examples), then the difference equation should follow a **step**, **OUT**, or **SAMPLE m** statement at the end of the DYNAMIC program segment.

(b) Track-hold Simulation

The difference equation

$$\mathbf{y} = \mathbf{y} + \text{swtch}(\text{ctrl}) * (\mathbf{x} - \mathbf{y}) \quad (2-20)$$

models a track-hold (sample-hold) circuit. The “continuous” difference-equation state variable \mathbf{y} tracks the input \mathbf{x} when the control variable **ctrl** is positive and holds its last value when **ctrl** is less than or equal to 0. Figure 2-8 illustrates the track-hold action.

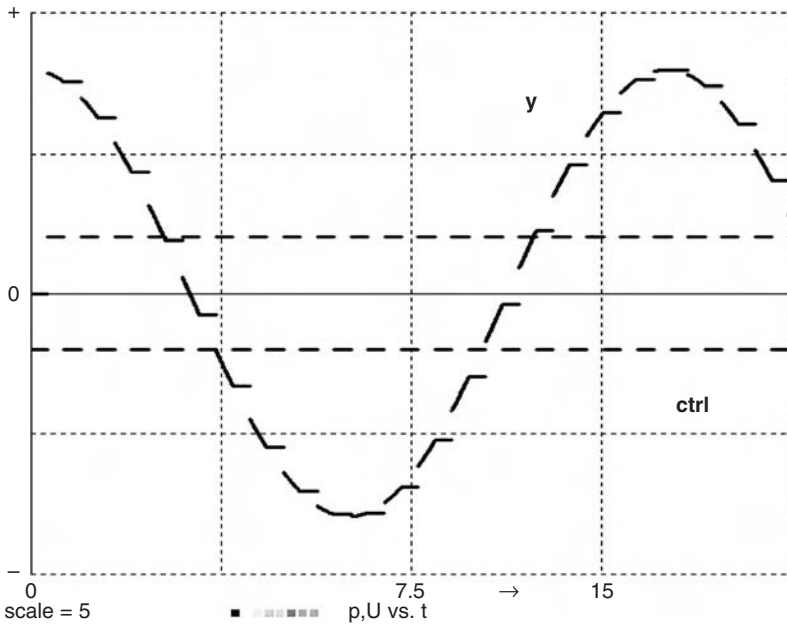


FIGURE 2-8. Track-hold operation modeled with the difference equation $\mathbf{y} = \mathbf{y} + \text{swtch}(\text{ctrl}) * (\mathbf{x} - \mathbf{y})$. The control waveform was obtained with the program of Figure 2-14.

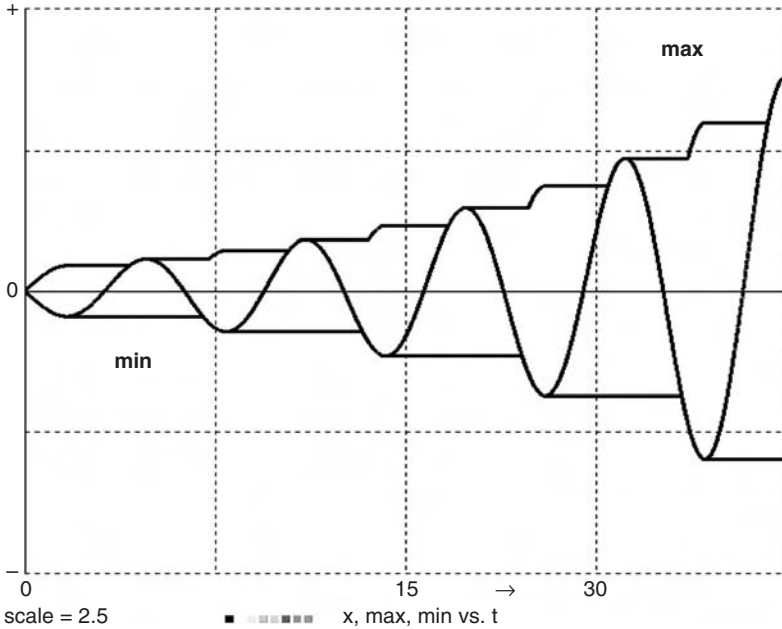


FIGURE 2-9. Maximum and minimum holding with the difference equations (2-21) and (2-22).

(c) Maximum- and Minimum-value Holding

The difference-equation state variable

$$\mathbf{max} = \mathbf{x} + \lim(\mathbf{max} - \mathbf{x}) \quad (2-21)$$

tracks and holds the largest past value of $\mathbf{x} = \mathbf{x}(t)$ (Fig. 2-9; see also Section 2-13; [3]). DESIRE automatically assigns **max** the initial value 0; since that would keep **max** from remembering negative values of \mathbf{x} , we initialize **max** with a large negative value such as $-1.0\text{E}+30$.

As also shown in Figure 2-9, the difference-equation state variable

$$\mathbf{min} = \mathbf{x} - \lim(\mathbf{x} - \mathbf{min}) \quad (2-22)$$

similarly holds the smallest past value of \mathbf{x} ; we initialize min with $1.0\text{E}+30$. An example in the book CD applies Eq. (2-21) to hold the largest past value of $|\mathbf{x}|$ for automatic display scaling [9].

(d) Simple Backlash and Hysteresis Models

The difference equation

$$\mathbf{y} = \mathbf{y} + \mathbf{a} * \text{deadz}((\mathbf{x} - \mathbf{y})/\mathbf{a}) \quad (2-23)$$

models the transfer characteristic of one-way simple backlash (e.g., gear backlash) from x to y (Fig. 2-10; [3]). We can use y to drive various continuous-function generators, for example,

$$z = \tanh(10 * y)$$

to obtain other transfer characteristics exhibiting hysteresis or memory of past input values (Fig. 2-11). Truly realistic hysteresis models, though, should be developed directly from physics; they are likely to involve differential equations as well as difference equations.

A different example, the difference equation

$$y = \text{deadc}(A * y - x) \quad (2-24)$$

produces the transfer characteristic of a deadspace comparator with hysteresis (Fig. 2-12). This has been used to model the operation of pairs of space-vehicle on-off vernier control rockets.⁹

(e) The Comparator with Hysteresis (Schmitt Trigger)

The most useful of our hysteresis-type difference equations

$$p = A * \text{sgn}(p - x) \quad (2-25)$$

directly models a comparator with regenerative feedback, the *Schmitt trigger* circuit widely used by electrical engineers (Fig. 2-13; [2,3,10]). The

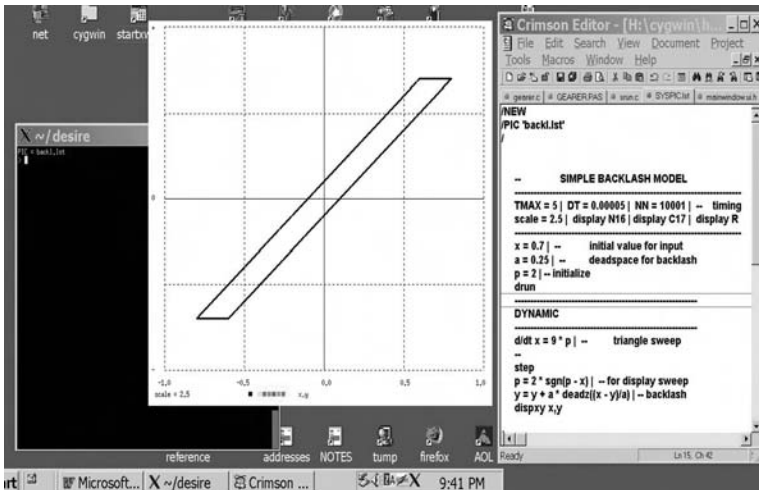


FIGURE 2-10. This Cygwin display shows a simple backlash transfer characteristic with a demonstration program sweeping x with the sawtooth waveform of Section 2-15.

⁹ Equation (2-24) corrects a printing error in Reference [4].

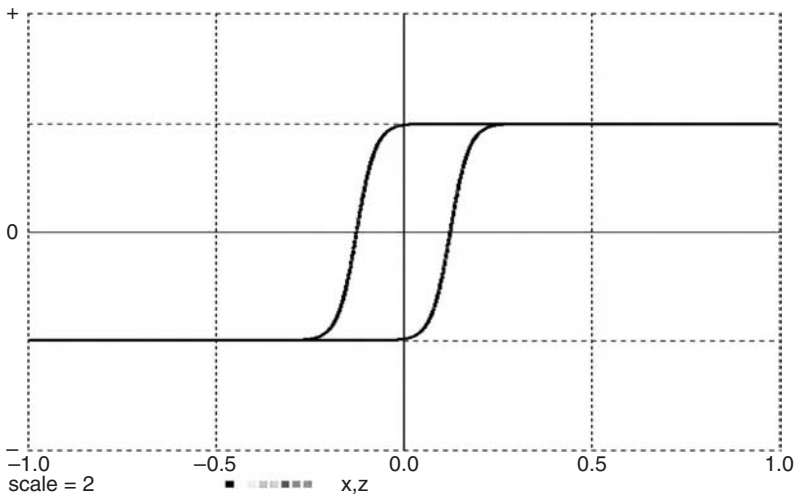


FIGURE 2-11. A simple hysteresis transfer characteristic.

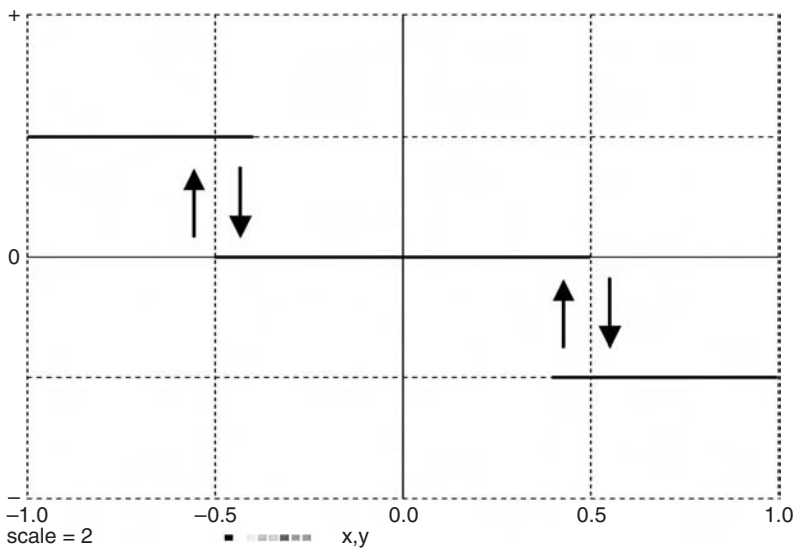


FIGURE 2-12. Transfer characteristic (y versus x) of a deadspace comparator with hysteresis.

difference-equation state variable p defaults to 0 but is usually initialized to $-A$ or $+A$. This modeling trick was already used with early fixed-point block-diagram simulation languages [2,3].

Simulated Schmitt triggers often replace deadspace comparators in control systems (Example 2.1), but perhaps their most useful application is to the generation of periodic signals (Section 2-17).

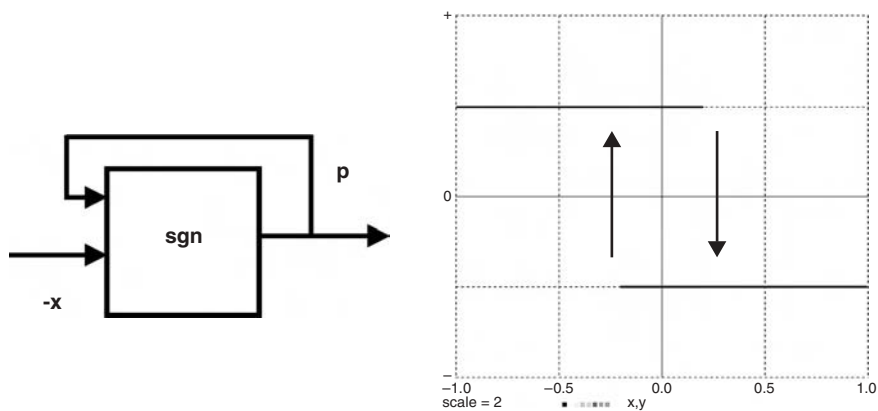


FIGURE 2-13. A comparator with regenerative feedback (Schmitt trigger) implemented with $p = A * \text{sgn}(p - x)$, and its transfer characteristic p versus x .

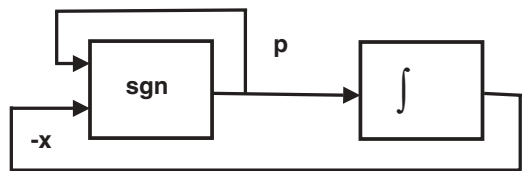


FIGURE 2-14. Integrator feedback around a Schmitt trigger model produces a useful signal generator. The resulting square waves $p(t)$ and triangle waves $x(t)$ are shown in Figure 2-15.

2-17. Signal Generators and Signal Modulation

Feeding the time-integrated output of a hardware or software Schmitt trigger back to the input (Fig. 2-14) recreates the classical Hewlett-Packard signal generator [2,3,10]. This is implemented with the simple program

```
TMAX = 5 | DT = 0.0001 | NN = 5000

-----

A = 0.22 | a = 4 | — signal parameters
x = 1 | p = 1 | — initialize
drun
```

DYNAMIC

$d/dt x = a * p$ | -- triangle waves

step

$p = \text{sgn}(p - x)$ | -- square waves (2-26)

The experiment protocol usually initializes the difference-equation state variable p and the differential-equation state variable x with $p = A$ and $x = -A$.

When $p = A$, the integrator output x increases until $-x$ overcomes the positive Schmitt-trigger bias $p = A$ in Eq. (2-25). p now switches to $-A$, and x decreases until it reaches the new trigger level $-A$. This process repeats and generates a square wave $p = p(t)$ and a triangle wave $x = x(t)$, both of amplitude A and frequency $a/(4 * A)$ (Fig. 2-15). Frequency resolution is determined by the switching-time resolution, that is, by the largest DT value used in the integration routine (Sections 2-10 and 2-11).

These periodic functions are useful as computer-generated test signals and control signals.¹⁰ An added assignment

$y = p * x$ (2-27)

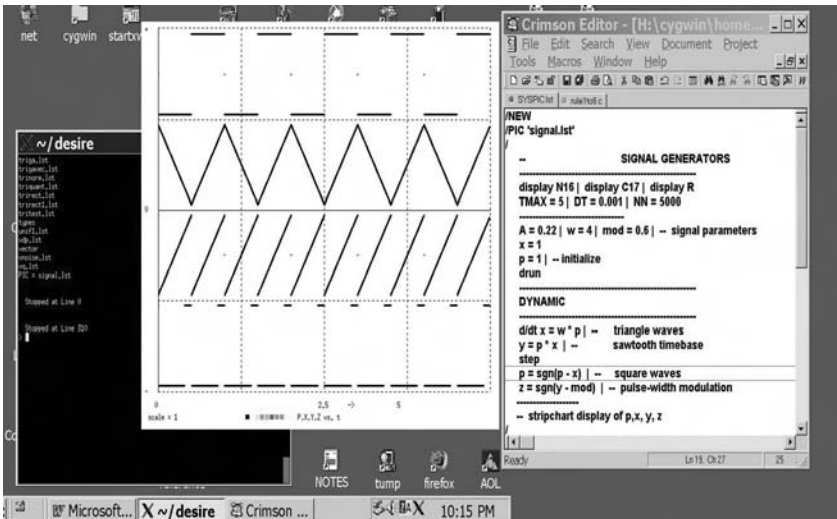


FIGURE 2-15. This Cygwin (Unix under Windows™) screen shows a terminal window, an editor window, and graphics demonstrating the signal-generator program in Section 2-17. The original display showed different curves in different colors.

¹⁰ We used the triangle wave $x(t)$ to sweep the input to all the function-generator displays shown in this chapter.

generates a sawtooth waveform y that sweeps between $-A$ and A with frequency $0.5 * a/A$. One can produce a large variety of more general periodic waveforms by feeding $p(t)$ or $y(t)$ to various function generators, as in

$$z = f(y) \quad (2-28)$$

$f(y)$ can be a library function, a user-defined function, or a table-lookup function.

We can frequency-modulate all these periodic waveforms by making the parameter a a variable. One can also add a variable bias $-mod$ to the sawtooth waveform y and send the result to a comparator whose output

$$z = \text{sgn}(y - \text{mod})$$

is then a train of pulse width-modulated pulses (Fig. 2-15). We note here that computer-generated sinusoidal signals $s = A * \sin(w * t + \text{phi})$ can also be amplitude-, frequency-, and/or phase-modulated by making the parameters a , w , and phi variable.

REFERENCES

1. G. F. Franklin, *Digital Control of Dynamic Systems*, Addison-Wesley, Reading, MA, 1990.
2. G. A. Korn and J.V. Wait, *Digital Continuous-System Simulation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
3. G. A. Korn, Tricks and treats: Nonlinear operations in digital simulation, *Math and Computers in Simulation*, **29**, 1987, pp. 129–143.
4. G. A. Korn, *Interactive Dynamic-System Simulation with Microsoft Windows*, Taylor and Francis, London, 1998.
5. F. E. Cellier and D. F. Rufer, Algorithm for the solution of initial-value problems, *Math and Computers in Simulation*, **20**, 1978, pp. 160–165.
6. F. Cellier and E. Kofman, *Continuous-System Simulation*, Springer, New York, 2006.
7. M. B. Carver, Efficient integration over discontinuities, *Math and Computers in Simulation*, **20**, 1978, pp. 190–196.
8. D. Ellison, Efficient automatic integration of ordinary differential equations with discontinuities, *Math and Computers in Simulation*, **23**, 1981, pp. 12–20.
9. C. W. Gear, Efficient step-size control for output and discontinuities, *Transactions SCS*, **1**, 1984, pp. 27–31.
10. G. A. Korn and T. M. Korn, *Electronic Analog and Hybrid Computers*, 2nd Ed., McGraw-Hill, New York, 1964.

3

Programs with Vector/Matrix Operations and Submodels

VECTOR ASSIGNMENTS AND VECTOR DIFFERENTIAL EQUATIONS

3-1. Arrays, Subscripted Variables, and State-variable Declarations

Array declarations such as

ARRAY $x[n]$ | **ARRAY** $A[n, m]$ or **ARRAY** $x[n], A[n, m]$

in DESIRE experiment-protocol scripts define one- and two-dimensional arrays (vectors¹ and matrices²) of subscripted real variables $x[1], x[2], \dots, x[n]$ and $A[i, k]$ ($i = 1, 2, \dots, n; k = 1, 2, \dots, m$). Note that vectors and matrices are much more than a shorthand notation—they are intuitively meaningful abstractions in many useful models (e.g., forces and velocity vectors).

¹ Our vectors are indeed vectors in the mathematical sense, since Section 3.1.2 provides a suitable definition of vector addition, and multiplication of vectors by scalars.

² An $n \times m$ matrix declared with **ARRAY** $A[n, m]$ has n rows and m columns. DESIRE can also declare arrays with more dimensions, but they are rarely used.

All subscripted variables (array elements) initially default to 0. Experiment-protocol scripts can “fill” arrays with assignments to subscripted variables, as in

```
A[19, 4] = 7.3   |   v[2] = a - 3 * b  
for i = 1 to n   |   x[i] = 20 * i   |   next
```

or from **data** lists or files and **read** assignments, as in

```
data 1.2, - 4, a + 4 * b, 7.882, ...   |   read v, A, ...
```

Once declared, vectors and matrices, and the resulting subscripted variables, can be freely used both in the experiment protocol³ and in DYNAMIC program segments. DYNAMIC segments can assign time-variable expressions to array elements. The program flags undeclared subscripted variables, vectors, or matrices as *undefined*.

Before subscripted variables **x[i]**, **y[i]**, ... or vectors **x**, **y**, ... can be used as state variables in differential equations (Section 1-2), the experiment-protocol script must declare one-dimensional state-variable arrays (state vectors) with a **STATE** declaration such as

```
STATE x[n], y[m], ...
```

Scalar state variables need not be declared, unless they are to be used in sub-models (Section 3-17) or in more than one DYNAMIC program segment.

3-2. Vector Operations in DYNAMIC Program Segments— The Vectorizing Compiler

(a) Vector Assignments and Vector Expressions

Assume that the experiment protocol has declared vectors **y1**, **y2**, **y3**, ... all of the same dimension **n**, say with

```
ARRAY y1[n], y2[n], y3[n], ... (3-1)
```

Then a vector assignment [1]

```
Vector y1 = g(t; y2, y3, ... ) (3-2a)
```

³ The detailed syntax of the script language is described in the DESIRE reference manual supplied in the book CD.

in a DYNAMIC program segment compiles automatically into n scalar assignments

$$\mathbf{y1}[i] = \mathbf{g}(\mathbf{t}; \mathbf{y2}[i], \mathbf{y3}[i], \dots) \quad (i = 1, 2, \dots, n) \quad (3-2b)$$

The simulation time variable is \mathbf{t} . $\mathbf{g}()$ stands for any expression that can be used in a scalar assignment. Such a vector expression may involve literal numbers, scalar parameters, parentheses, library functions, user-defined functions, or table-lookup functions. An error is returned when one tries to combine vectors with unequal dimensions.

For example, if \mathbf{y} , \mathbf{u} , \mathbf{v} , and \mathbf{z} are n -dimensional vectors, then

$$\text{Vector } \mathbf{y} = (\mathbf{1} - \mathbf{v}) * (\cos(\alpha * \mathbf{z} * \mathbf{t}) + 3 * \mathbf{u})$$

compiles into

$$\mathbf{y}[i] = (\mathbf{1} - \mathbf{v}[i]) * (\cos(\alpha * \mathbf{z}[i] * \mathbf{t}) + 3 * \mathbf{u}[i]) \quad (i = 1, 2, \dots, n)$$

Note that the expression $\mathbf{g}()$ is the same for all n vector components $\mathbf{y1}[i]$. The DESIRE compiler reads the vector dimension n from the array data structure. The code for n successive vector components is then generated by a compiler loop. Each pass through this loop compiles all the operations for the expression $\mathbf{g}(\mathbf{y2}[i], \mathbf{y3}[i], \dots)$ and then automatically increments the vector index i . The resulting “vectorized” code is fast, for there is no runtime loop overhead.

A DYNAMIC program segment can have multiple vector assignments with the same or different dimensions.

(b) Vector Differential Equations

Assume that the experiment protocol has declared the n -dimensional arrays (3.1) and has also declared an n -dimensional state vector \mathbf{x} with

STATE $\mathbf{x}[n]$

Then a vector differential equation (vector state equation)

$$\text{Vectr } d/dt \mathbf{x} = \mathbf{f}(\mathbf{t}; \mathbf{x}, \mathbf{y1}, \mathbf{y2}, \dots) \quad (3-3a)$$

in a DYNAMIC program segment compiles automatically into n scalar differential equations

$$d/dt \mathbf{x}[i] = \mathbf{f}(\mathbf{t}; \mathbf{x}[i], \mathbf{y1}[i], \mathbf{y2}[i], \dots) \quad (i = 1, 2, \dots, n) \quad (3.3b)$$

$\mathbf{f}()$ represents an arbitrary vector expression, just as in Section 3-2a. The initial values of the subscripted state variables $\mathbf{x}[i]$ default to 0 unless the experiment protocol assigns other values. After a simulation run, initial values of

all differential-equation state variables can be reset by **reset** and **drunr** statements in the experiment-protocol script.

A DYNAMIC program segment may contain any number of vector assignments and vector differential equations together with scalar assignments and/or differential equations. Different vector-assignment targets and state vectors can have different dimensions. Scalar expressions can also contain explicit subscripted variables, provided that their arrays have been declared.

(c) *Vectorization and Model Replication—Significant Applications*

A given system of **n**-dimensional vector assignments and **n**-dimensional vector differential equations, say

Vector $y1 = g1(t; x1, x2; a, \alpha)$
Vector $y2 = g2(t; x1, x2, y1; \beta)$
Vectr $d/dt\ x1 = f1(t; x1, x2; y1, y2; b, c)$
Vectr $d/dt\ x2 = f2(t; x1, x2; \gamma)$

is compiled into corresponding sets of **n** scalar operations

$y1[i] = g1(t; x1[i], x2[i]; a[i], \alpha) \quad (i = 1, 2, \dots, n)$
 $y2[i] = g2(t; x1[i], x2[i], y1[i]; \beta) \quad (i = 1, 2, \dots, n)$
 $d/d\ x1[i] = f1(t; x1[i], x2[i]; y1[i], y2[i]; b[i], c[i]) \quad (i = 1, 2, \dots, n)$
 $d/dt\ x2[i] = f2(t; x1[i], x2[i]; \gamma) \quad (i = 1, 2, \dots, n)$

in that order. The compiler effectively creates **n replicated models**.⁴ These models have different parameter combinations **a[i]**, **b[i]**, **c[i]** defined by the parameter vectors **a**, **b**, and **c**, but all **n** replicated models share the features represented by the scalar parameters or variables **alpha**, **beta**, and **gamma**.⁵

Vectorization allows one to exercise a possibly large number of models in a single simulation run. Applications of this extraordinarily powerful simulation technique are the main topic of this book. Specifically,

- Vectorized parameter-influence studies simulate replicated models with different parameter values (Sections 4-2 and 4-3).
- Vectorized Monte Carlo simulation computes statistics on samples of models with random parameters or noise inputs (Sections 4-7 to 4-10; Chapter 5).

⁴ Note that successive operations for different models are interleaved in the computer memory.

⁵ Scalar quantities are common to all the replicated models and must, therefore, not depend in any way on the replicated-model vectors. That said, scalars can be defined by the experiment protocol, by DYNAMIC-segment assignments, or even by differential equations.

- Neural-network simulations can replicate different neuron models (Chapter 6.).
- The Method of Lines represents suitable partial differential equations as sets of ordinary differential equations (Sections 7-10 to 7-14).
- Map-based agroecology simulations replicate models of crop growth or species competition at different points of a landscape (Sections 7-15 and 7-16).

3-3. Matrix-vector Products in Vector Expressions

(a) Definition

Any n -dimensional vector in the vector expressions \mathbf{f} or \mathbf{g} in Sections 3-1 and 3-2, say $\mathbf{y2}$, can be a matrix-vector product $\mathbf{A} * \mathbf{v}$. Here, \mathbf{A} is a rectangular $n \times m$ matrix, and \mathbf{v} is an m -dimensional vector,⁶ both declared in the experiment protocol. Expressions for \mathbf{A} or \mathbf{v} cannot be substituted, but \mathbf{A} and \mathbf{v} can be defined by preceding assignments. Nonconformable matrix-vector products are automatically rejected with an error message. More specifically, a matrix-vector-product assignment such as

Vector $\mathbf{y} = \tanh(\mathbf{A} * \mathbf{v})$

compiles into n scalar assignments

$$\mathbf{y}[i] = \tanh\left(\sum_{k=1}^m \mathbf{A}[i,k] * \mathbf{v}[k]\right) \quad (i = 1, 2, \dots, n)$$

A vector \mathbf{v} used in a matrix-vector product $\mathbf{A} * \mathbf{v}$ must be a simple vector or state vector; it cannot be a vector expression, matrix-vector product, or index-shifted vector (Section 3-6). As we already noted, though, one can assign an m -dimensional vector expression to \mathbf{v} with a preceding vector assignment. In particular, cascaded linear transformations

Vector $\mathbf{z} = \mathbf{B} * \mathbf{v}$ | Vector $\mathbf{y} = \mathbf{A} * \mathbf{z}$

effectively multiply \mathbf{v} by the matrix product \mathbf{AB} of two appropriately dimensioned matrices.

⁶ If \mathbf{v} is identical with assignment target of a **Vector** (or **Vectr delta**, Section 3.1.4) operation, the program returns an “illegal recursion” error. For example, **Vector $\mathbf{x} = \mathbf{A} * \mathbf{x}$** is illegal and must be replaced with **Vector $\mathbf{z} = \mathbf{A} * \mathbf{v}$** followed by **Vector $\mathbf{y} = \mathbf{z}$** , just as in Fortran or C. There is no such problem in **Vectr d/dt** operations, since the compiler assigns a hidden intermediate variable for each derivative.

For matrix-vector products written as $\mathbf{A} \% * \mathbf{x}$, DESIRE transposes the matrix \mathbf{A} . This is useful in neural-network applications (Section 6-12).

(b) A Simple Example: Resonating Oscillators

The following model is typical of a large class of mass–spring systems. The differential-equation system

$$\begin{aligned} d/dt \, x1 &= x1dot & | & \quad d/dt \, x1dot = -ww * x1 - k * (x1 - x2) \\ d/dt \, x2 &= x2dot & | & \quad d/dt \, x2dot = -ww * x2 - k * (x2 - x1) - r * x2dot \end{aligned} \quad (3.4)$$

models a pair of harmonic oscillators coupled by a spring. The first oscillator is undamped, and the second oscillator has viscous damping. When the system is started with an initial displacement $\mathbf{x}[1] = 0.5$, the second oscillator resonates with the motion of the first oscillator; the damping in the second oscillator eventually dissipates the energy of both systems (Fig. 3-1).

The simulation program in Figure 3-1 models the same fourth-order system with one vector differential equation. The experiment-protocol script declares a four-dimensional state vector \mathbf{x} and a 4×4 matrix \mathbf{A} with

STATE $\mathbf{x}[4]$ | ARRAY $\mathbf{A}[4,4]$

We then represent the state variables $\mathbf{x1}$, $\mathbf{x1dot}$, $\mathbf{x2}$, $\mathbf{x2dot}$, in that order, by state-vector components (subscripted variables) $\mathbf{x}[1]$, $\mathbf{x}[2]$, $\mathbf{x}[3]$, $\mathbf{x}[4]$. The 4×4 matrix

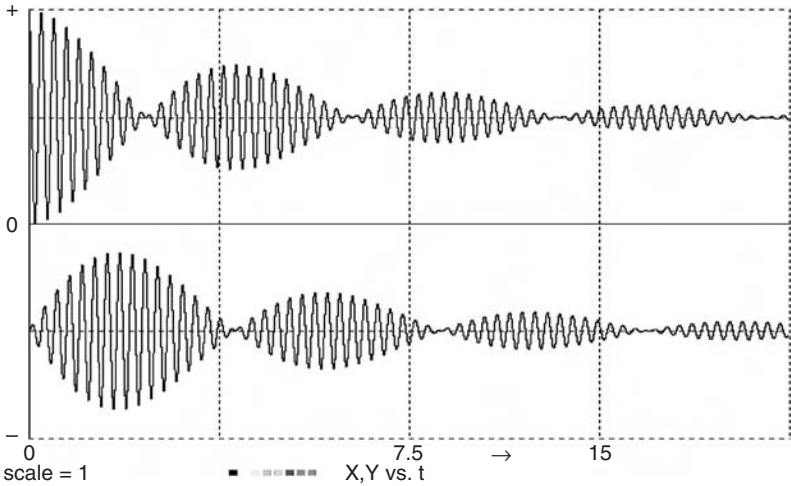
$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -(ww+k) & -k & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -k & -(ww+k) & 0 & -r \end{bmatrix}$$

is filled with a data/read assignment

data 0, 0, 1, 0; 0, 0, 0, 1; - (ww + k), - k, 0, 0; - k, - (ww + k), 0, - r | read A

3-4. Vector Sampled-data Assignments and Vector Difference Equations

Subscripted variables, and thus also vectors and matrices, can be sampled-data variables as well as “continuous” variables, so that there can be vector assignments, including vector difference equations, and updating assignments, in



```
--
                                RESONATING OSCILLATORS
-----
TMAX = 15 | DT = 0.00001 | NN = 100000
--
ww = 600 | --      circular frequency
k = 40   | --      coupling coefficient
r = 0.7  | --      damping coefficient
--
STATE x[4] | ARRAY A[4,4]
data 0, 0, 1, 0; 0, 0, 0, 1; - (ww + k), - k, 0, 0; - k, - (ww + k), 0, - r | read A
--
x[1] = 0.5 | --      initial value
drun
-----
DYNAMIC
-----
Vectr d/dt x = A * x
```

FIGURE 3-1. Matrix-vector form of the resonating-oscillator simulation.

DYNAMIC-segment sampled-data sections following an **OUT** or **SAMPLE m** statement, as in the case of scalars (Section 2-1). Another way to program vector difference equations is in the “incremental” form

Vectr delta q = vector expression

which is equivalent to

Vector q = q + vector expression

q is a difference-equation state vector (see also Section 2-2). Note that the initial values of all array components not explicitly specified in the experiment protocol default to 0. This is true for all subscripted variables, not just state variables.

3-5. Sorting Vector and Subscripted-variable Assignments

Vector defined-variable assignments for differential equations or difference equations must be sorted as in Sections 1-9 and 2-1, but now sort errors cannot return “undefined variable” messages, since all arrays are predefined. Simple models can be sorted by inspection. One may also be able to sort replicated (vectorized) models in scalar form before adding their **Vectr d/dt** and **Vector** prefixes.

Explicit assignments to subscripted variables, say

$$\text{d/dt } x[2] = -x[3] \qquad y[n] = a * \sin(t) + b$$

are normally only used to “amend” a preceding **Vectr d/dt** or **Vector** assignment for selected index values, as in Section 7-6b and Table 7-1. That poses no sorting problems.

MORE VECTOR OPERATIONS

3-6. Index-shifted Vectors

Given a properly declared vector $\mathbf{v} \equiv (\mathbf{v}[1], \mathbf{v}[2], \dots, \mathbf{v}[n])$, *index-shifted* versions $\mathbf{v}\{\mathbf{k}\}$ of \mathbf{v} can be introduced in n -dimensional vector expressions, but not in matrix-vector products or **DOT** products. The index shift \mathbf{k} is a rounded scalar expression computed at compile time.

When a vector expression containing $\mathbf{v}\{\mathbf{k}\}$ is compiled, $\mathbf{v}\{\mathbf{k}\}$ contributes index-shifted vector components $\mathbf{v}[\mathbf{i} + \mathbf{k}]$ wherever \mathbf{v} would contribute vector components $\mathbf{v}[\mathbf{i}]$. Thus, if $\mathbf{y1}, \mathbf{y2}, \dots$ are n -dimensional vectors,

$$\text{Vector } \mathbf{y1} = \mathbf{g}(t; \mathbf{y2}, \mathbf{y3}\{\mathbf{k}\}, \dots) \qquad (3-5a)$$

compiles into the n scalar assignments

$$\mathbf{y1}[\mathbf{i}] = \mathbf{g}(t; \mathbf{y2}[\mathbf{i}], \mathbf{y3}[\mathbf{i} + \mathbf{k}], \dots) \quad (\mathbf{i} = 1, 2, \dots, n) \qquad (3-5b)$$

where the compiler sets $\mathbf{v}[\mathbf{i} + \mathbf{k}] = 0$ for $\mathbf{i} + \mathbf{k} < 0$ or $\mathbf{i} + \mathbf{k} > n$.⁷

⁷ An index-shifted vector \mathbf{x} appearing in a **Vector** or **Vectr delta** assignment must not be identical with the assignment target \mathbf{v} when the index shift is positive. In case it is, an illegal recursion is caused and an error message returned, since the system fills vector arrays starting with high index values. There is no such restriction for **Vectr d/dt** operations.

Vector-shift operations neatly implement relations between vector components with different indices. This has many interesting and useful modeling applications, such as

- shift registers, time delays, pseudorandom-noise generators, and digital signal processing,
- neural-network layers with memory and predictor networks (Section 6-22),
- partial differential equations (Section 7-11),
- fuzzy-logic membership functions (Section 7-7).

Specifically, vector-component values can be shifted along a vector array, and also successive samples of a scalar function $\mathbf{s}(\mathbf{t})$ of the simulation time \mathbf{t} can be shifted into and out of a vector array (Section 6-22).

3-7. Sums, DOT Products, and Vector Norms

(a) Sums and DOT Products

DESIRE **DOT** products assign inner products of vectors to scalar variables. In both DYNAMIC program segments and experiment-protocol scripts,

$$\text{DOT } \mathbf{xsum} = \mathbf{x} * \mathbf{1} \quad \text{assigns} \quad \sum_{k=1}^n \mathbf{x}[k] \quad \text{to } \mathbf{xsum}$$

$$\text{DOT } \mathbf{p} = \mathbf{x} * \mathbf{1} \quad \text{assigns} \quad \sum_{k=1}^n \mathbf{x}[k] \mathbf{y}[k] \quad \text{to } \mathbf{p}$$

Compiled sums and **DOT** products do not incur any summation-loop overhead (loop-unrolling compilation).

The vectors \mathbf{x} and \mathbf{y} in a **DOT** operation must not be vector expressions or index-shifted vectors. But \mathbf{y} can be a matrix-vector product $\mathbf{A} * \mathbf{v}$ or $\mathbf{A}\% * \mathbf{v}$ (Section 3-3), so that bilinear forms $\mathbf{x} * \mathbf{A} * \mathbf{y}$ and quadratic forms $\mathbf{x} * \mathbf{A} * \mathbf{x}$ can be neatly evaluated. DESIRE automatically rejects nonconformable products with an error message.

DYNAMIC program segments also accept sums of **DOT** products, for example,

$$\text{DOT } \mathbf{p} = \mathbf{p} * \mathbf{q} + \mathbf{r} * \mathbf{s} + \mathbf{x} * \mathbf{A} * \mathbf{y} + \mathbf{z} * \mathbf{1}$$

(b) Euclidean, Taxicab, and Hamming Norms

DOT assignments efficiently compute squared vector norms, which are often needed as error measures in statistics and optimization studies. In particular,

$$\text{DOT } \mathbf{xnormsq} = \mathbf{x} * \mathbf{x}$$

produces the squared Euclidean norm

$$\mathbf{xnormsq} = \|\mathbf{x}\|^2 = \sum_{k=1}^n \mathbf{x}^2[k]$$

of a vector \mathbf{x} . The Euclidean distance between two vectors \mathbf{x} , \mathbf{y} is the norm $\|\mathbf{x} - \mathbf{y}\|$ of their difference. Thus

$$\mathbf{Vector\ e = x - y} \quad | \quad \mathbf{DOT\ enormsq = e * e}$$

produces the useful error measure

$$\mathbf{enormsq} = \sum_{k=1}^n (\mathbf{x}[k] - \mathbf{y}[k])^2$$

The sums of scalar functions can be computed conveniently as in

$$\mathbf{S = exp(x[1]) + exp(x[2]) + exp(x[2]) + \dots + exp(x[n])}$$

with

$$\mathbf{Vector\ y = exp(x)} \quad | \quad \mathbf{DOT\ S = y * 1}$$

In particular,

$$\mathbf{Vector\ xa = abs(x)} \quad | \quad \mathbf{DOT\ xanorm = xa * 1}$$

generates the *taxicab norm* (city-block norm) $\mathbf{anorm} = |(\mathbf{x}[1])| + |(\mathbf{x}[2])| + \dots$ of a vector \mathbf{x} . The taxicab norm of a vector difference (taxicab distance, as in a city with rectangular blocks) is another useful error measure.

If all components $\mathbf{x[i]}$ of a vector \mathbf{x} equal 0 or 1, the taxicab norm reduces to the *Hamming norm*, which simply counts the nonzero elements. The Hamming distance $\|\mathbf{x} - \mathbf{y}\|$ between two such vectors is the count of corresponding element pairs that differ.

3-8. Maximum/Minimum Selection and Masking

(a) Maximum/Minimum Selection

The vector assignment

$$\mathbf{Vector\ y^{\wedge} = vector\ expression}$$

computes the vector produced by $\mathbf{Vector\ y = vector\ expression}$ and then sets all but its largest component to 0. Such vectors are particularly useful as

pattern selectors in neural-network simulations (Section 6-4). To find the value **y_{max}** of the largest vector component of *vector expression*, use

Vector y[^] = vector expression | DOT y_{max} = y * 1

The index **l** of the largest vector component **y[i]** can be determined with a small loop in the experiment-protocol script:

i = 0 | repeat | i = i + 1 | until y[i] <> 0 | l = i

The smallest vector component of *vector expression* is, of course, minus the largest component of $- \textit{vector expression}$. Maximum or minimum selection is useful in parameter-influence studies and optimization studies (Section 4-3d). Note that these operations apply neatly to arrays created by vector equivalences (Section 3-11).

(b) Masking Vector Expressions

Vector expressions used with **Vector** and **Vectr d/dt** operations (and also with **Vectr delta** operations, Section 3-4) can be masked with an **n**-dimensional mask vector **vv**, as in

Vector x = [vv] vector expression

Vectr d/dt x = [vv] vector expression

The **i**th component of a masked vector expression is set to 0 for all values of the index **i** such that **vv[i] ≠ 0**. Mask vectors **vv** are set up by the experiment-protocol program and do not change in the course of a simulation run. Vector masking has been used to “prune” neuron layers in neural-network simulations.

MATRIX OPERATIONS

3-9. Matrix Operations in Experiment-protocol Scripts

DESIRE models use matrices in matrix-vector products (Section 3-3), and also as row-pattern matrices in neural-network studies (Section 6-5b). We saw in Section 3-1 how experiment-protocol scripts declare and “fill” matrix arrays. For convenience, DESIRE experiment-protocol scripts can also produce square null and unit matrices, transposed and inverse square matrices, and products of square matrices for use later in the program. After the square matrices **A**, **B**, **C**, ... are declared,

MATRIX A = 0

resets all **A[i, k] = 0**

MATRIX A = 1

produces a unit matrix **A** (1s along diagonal)

MATRIX B = \$ln(A) makes **B** the matrix inverse of **A** (if it exists)
MATRIX B = A% makes **B** the matrix transpose of **A**
MATRIX D = a * A * B * C * ... produces a matrix product **D** (**a** is an optional scalar)

These assignments return error messages if matrices are not square or unconformable, or if an inverse does not exist. As noted, for properly dimensioned rectangular matrices **A**, **B**

MATRIX B = A% makes **B** the transpose of **A** ($b[i, k] = a[k, i]$ for all i, k)

Nonconformable matrices **A**, **B** are again rejected with an error message.

3-10. Matrix Assignments and Difference Equations in DYNAMIC Program Segments

DYNAMIC program segments can manipulate matrices declared in the experiment protocol with matrix assignments

MATRIX W = matrix expression

Matrix expressions are functions of scalars **a**, **b**, ..., vectors **u**, **v**, ..., and/or matrices **A**, **B**, ..., which can be constants or variables. Some examples are

MATRIX W = a * A + b	($W[i, k] = a * A[i, k] + b$)
MATRIX W = a * A + b * B	($W[i, k] = a * A[i, k] + b * B[i, k]$)
MATRIX W = recip(A)	($W[i, k] = 1/A[i, k]$)
MATRIX W = sin(A)	($W[i, k] = \sin(A[i, k])$)
MATRIX W = u * v	($W[i, k] = u[i] v[k]$)
MATRIX W = u & v	($W[i, k] = \min\{u[i], v[k]\}$)

The syntax of more general matrix expressions is defined in the DESIRE reference manual. Matrices can, moreover, be manipulated as equivalent vectors (Section 3-11).

Matrix difference equations are used mainly to modify matrix-vector products $W * x$ in optimization studies (control systems, statistical regression, model matching, and neural networks). In particular, the matrix difference equation

DELTA W = matrix expression

is equivalent to

MATRIX W = W + matrix expression

The resulting matrix elements **W[i, k]** are difference-equation state variables (Section 2-2). Their initial values default to zero unless otherwise specified by the experiment protocol. They are not reset by **reset** or **drunr** statements. The precautions of Section 2-2 apply.

3-11. Vector and Matrix Operations using Equivalent Vectors

DESIRE experiment protocol scripts can use two very useful equivalence declarations similar to those in Fortran. In particular, the modified **ARRAY** declaration

ARRAY x1[n1] + x2[n2] + ... = x, ...

declares concatenated subvectors **x1, x2, ...** together with a vector **x** of dimension **n1 + n2 + ...** whose elements overlay the subvectors **x1, x2, ...**, starting with **x1**. One can then access, say, **x2[3]** also as **x[n1+3]**. Subvectors are particularly useful in neural-network simulations (Section 6-2).

The second type of equivalence declaration

ARRAY V[n, m] = v

allows one to access a two-dimensional array and its elements both as an **n × m** matrix **V** and as a vector **v** with dimension **nm**. Then equivalent vector expressions with the convenient **Vector** and **Vectr d/dt** operations to relate and modify matrices can be used. This technique can often (not always) replace matrix assignments. Applications include image processing, fuzzy-logic models (Section 7-7), and landscape modeling (Section 7-15).

Note that both concatenated subvectors and equivalent array vectors allow identification of maximum and minimum elements of large arrays by the method of Section 3-8.

VECTORS IN PHYSICS AND CONTROL-SYSTEM PROBLEMS

3-12. Vectors in Physics Problems

Vectors such as forces or velocities are not just useful shorthand notations for multiple equations; they are intuitively meaningful abstractions. Many relations used in physics problems are most easily understood when we represent them in vector form, for example,

Vectr d/dt position = velocity | Vectr d/dt velocity = force/mass

However, to obtain numerical results such as trajectory plots, it is usually necessary to specify vector components and initial values as scalar subscripted variables.

3-13. Simulation of a Nuclear Reactor

The program in Figure 3-2 shows a compact vector model of the chain reaction in a nuclear reactor. D. Hetrick's classical textbook problem [1-3] lumps the entire reactor into a single core region and neglects chain-reaction poisoning by reaction products such as xenon. The state variables are the normalized chain-reaction power output **enp** (proportional to neutron density), the reactor temperature **temptr**, and six normalized precursor-product densities **d[1]**, **d[2]**, ..., **d[6]**. Our compact vector model collects the state variables **d[i]** into a six-dimensional state vector **d**.

When the control-rod input **b * t** increases the reactivity **r**, the chain reaction increases **enp** dramatically. In the educational TRIGA reactor, the resulting increase in the reactor temperature in turn reduces the reactivity **r**, so that a short and safe power pulse results (Fig. 3-2b).

3-14. Linear Transformations and Rotation Matrices

Simple vector assignments such as **Vector y = A * x** conveniently implement linear operations on vectors, such as rotations. Note that **y = A * x** can represent the result of rotating the vector **x** into a new position, or **y** may be a representation of **x** in a rotated coordinate system.

The rotation of a plane vector **x** \equiv (**x[1]**, **x[2]**) into the vector **y** \equiv (**y[1]**, **y[2]**) can be programmed with two scalar defined-variable assignments

$$\begin{aligned} y[1] &= x[1] * \cos(\text{theta}) - x[2] * \sin(\text{theta}) \\ y[2] &= x[1] * \sin(\text{theta}) + x[2] * \cos(\text{theta}) \end{aligned}$$

Instead, a two-dimensional rotation matrix **A** with **ARRAY A[2, 2]** can be declared in the experiment protocol, and then possibly time-variable elements **A[i, k]** of **A** are specified in a DYNAMIC program segment:

$$\begin{array}{l|l} A[1,1] = \cos(\text{theta}) & A[1,2] = -\sin(\text{theta}) \\ A[2,1] = \sin(\text{theta}) & A[2,2] = \cos(\text{theta}) \end{array}$$

The rotation can now be modeled with

$$\text{Vector } y = A * x$$

```

--          Vector Model of the TRIGA Pulsed Nuclear Reactor
-----
--  r is the reactivity in dollars (we change r with time)
--  enp is the power in mw (proportional to neutron density)
--  en0 is the initial power in mw
--  en = enp/en0 is the normalized power (initial value is 1.0)
--  d[i] (i = 1, 2, ..., 6) are normalized precursor-product densities
--  bol = beta/l, lambda = al[i] (i = 1, 2, ..., 6)
--  f[i] (i = 1, 2, ..., 6) are normalized delayed neutron fractions
--  alf is the temperature coeff. of reactivity (dollar/cdeg)
--  ak is the reciprocal heat capacity (cdeg/mj)
-----
display N1 | display C8 | display Q | scale = 200
TMAX= 0.2 | NN = 5001 | DT = 0.00001
-----
a = 2.0 | b = 0.0 | bol = 140.0
alf = 0.016 | ak = 12.5 | gamma = 0.0267
en0 = 0.001
-----
STATE d[6] | ARRAY al[6], f[6]
--          fill the al and f arrays
data 0.0124, 0.0305, 0.111, 0.301, 1.14, 3.01 | read al
data 0.033, 0.219, 0.196, 0.395, 0.115, 0.042 | read f
--
data 1, 1, 1, 1, 1, 1 | read d | -- initial values of
--          precursor densities
temprt = 0 | en = 1.0 | --          initial values
drun
-----
DYNAMIC
-----
r = a + b * t - alf * temprt | -- b * t is a control-rod input
enp = en0 * en
DOT sum = f * d | --          note DOT product!
endot = bol * ((r - 1.0) * en + sum)
omega = endot/en | enlog = 0.4342945 * ln(en)
-----
d/dt temprt = ak * enp - gamma * temprt
d/dt en = endot
Vectr d/dt d = al * (en - d)
-----          offset display
ENP = enp - scale | dispt ENP

```

FIGURE 3-2a. Simulation program for a nuclear-reactor model using vector operations. **temprt** is the reactor temperature.

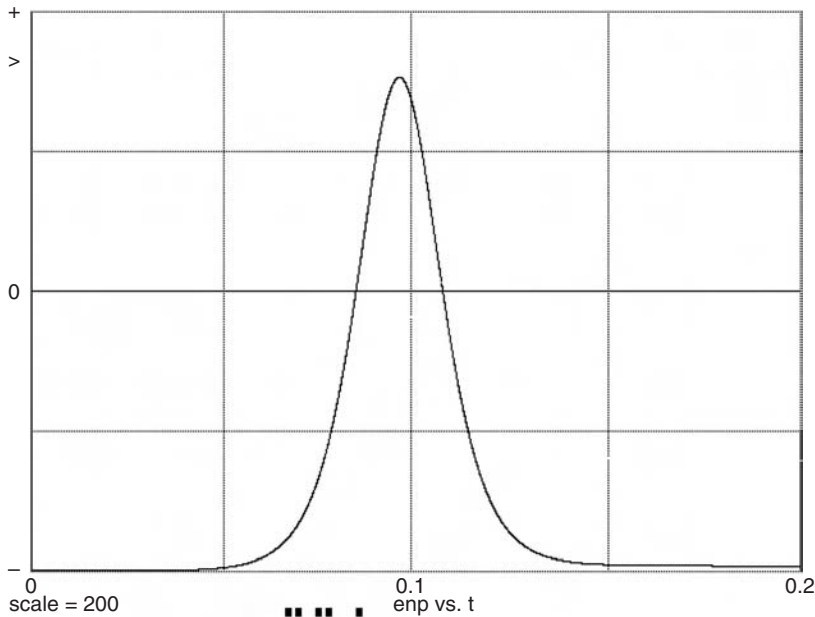


FIGURE 3-2b. Time-history plot of the reactor heat output **enp** generated by the program of Figure 3.2a. When the control increases the reactivity **r**, the chain reaction raises the reactor temperature. In the educational TRIGA reactor, this in turn reduces the reactivity, so that a short and safe power pulse results (based on Reference [1]).

The rotation matrix **A** representing our plane rotation is a useful abstraction; this becomes evident when we want to rotate several vectors **x1**, **x2**, ... through the same angle **theta**:

$$\text{Vector } y1 = A * x1 \quad | \quad \text{Vector } y2 = A * x2 \quad | \quad \dots\dots\dots$$

Three-dimensional rotation matrices are useful in flight simulations.

3-15. State-equation Models for Linear Control Systems

Modern textbooks [4] describe linear control systems by vector equations, which we represent in the computer-readable form

$$\text{Vectr } d/dt \, x = A * x + B * u$$

$$\text{Vector } y = C * x + D * u$$

x \equiv (**x1**, **x2**, ...) is a vector of state variables, and **u** and **y** are vectors of system input and output variables. The matrices **A**, **B**, **C**, and **D** define the plant and

controller and can be functions of the time t . Linear sampled-data control systems can be similarly described with vector sampled-data assignments [4].

USER-DEFINED FUNCTIONS AND SUBMODELS

DESIRE experiment-protocol scripts can define new functions and submodels as reusable language extensions. In subsequent DYNAMIC program segments, the DESIRE compiler invokes these subprograms as fast inline code without runtime function-call/return overhead.

Similar to vectors, user-defined functions and submodels are more than shorthand notations. They can be meaningful abstractions that make a simulation model much easier to understand, not just easier to program. Function and submodel definitions can be collected in library files for reuse.

3-16. User-defined Functions

Experiment-protocol scripts can create user-defined functions with **FUNCTION** declarations such as

FUNCTION abs2d(u\$, v\$) = sqrt(u\$^2 + v\$^2)

Once declared, the new function can be invoked in the experiment protocol or in a DYNAMIC program segment, say, with

RR = abs2d(x, y)

which would be exactly equivalent to the assignment **RR = sqrt(x^2 + y^2)**. DESIRE returns an error when declaration and invocation arguments do not match.

A function definition must fit one program line, but one should remember that a program line can be extended into another line on the display or listing. We marked the dummy arguments **u\$**, **v\$** with dollar signs so that they can be recognized easily, but this is not necessary. Dummy arguments must not be subscripted. Dummy-argument names are “protected” to prevent “side effects”. This means that any attempt to use their names after the function definition produces an error message. Function definitions may include constant parameters and also variables other than the dummy arguments.

An invocation argument can be any expression legal in the invocation context. Such expressions can include literals and subscripted variables. In experiment-protocol scripts, invocation arguments can be previously declared complex numbers or integers as well as real numbers. In DYNAMIC program segments, invocation arguments can be real or vector expressions.

TABLE 3-1. Some User-defined Functions*

```

FUNCTION max(x$, y$) = x$ + lim(y$ - x$)
FUNCTION min(x$, y$) = x$ - lim(x$ - y$)
FUNCTION cotan(x$) = cos(x$)/sin(x$)
FUNCTION asat(x$, alpha$) = alpha$ * sat(x$/alpha$) (alpha$ > 0)
FUNCTION bound(x$, alpha$, beta$)
    = lim(x$ - alpha$) - lim(x$ - beta$) + alpha$ (alpha$ < beta$)
FUNCTION relay(ctrl$, a$, b$) = b$ + (a$ - b$) * swtch(ctrl$)
FUNCTION tpulse(alpha$, beta$)
    = swtch(t - alpha$) - swtch(t - beta$) (alpha$ < beta$)

```

* See also Chapter 2 and Ref. 3.

User-defined functions can be collected in library files for reuse (Table 3-1). Here are some useful examples based on Section 2-13:

```

FUNCTION max(aa, bb) = aa + lim(bb - aa)
FUNCTION tpulse(aa, bb) = swtch(t - aa) - swtch(t - bb) (aa < bb)

```

FUNCTION definitions may be nested, that is, they can contain previously defined functions. But recursive definitions such as

```

FUNCTION f1(x) = f1(x) + 1

```

or

```

FUNCTION f1(x) = f2(1) + x | FUNCTION f2(y) = f1(y+1)

```

and also recursive function calls, as in

```

FUNCTION incr(x) = x + 1 | q = incr(incr(incr(y)))

```

are illegal.

3-17. Submodels

(a) Submodel Declaration and Invocation

Submodels defined in the experiment protocol can be invoked in DYNAMIC program segments to generate frequently used defined-variable operations and/or differential-equation systems in a single invocation line. Sections 6-12 and 7-12 illustrate applications of submodels.

Submodels must be declared in the experiment-protocol script before they are invoked in a DYNAMIC program segment. For example,

```
SUBMODEL quad(x$, y$, ydot$, a$, b$)
  d/dt y$ = ydot$
  d/dt ydot$ = x$ - a$ * y$ - b$ * ydot$
end
```

defines a differential-equation submodel representing a mass restrained by a spring and viscous friction. Once a submodel is declared, it can be invoked in any DYNAMIC program segment with appropriate variable or parameter names substituted for each dummy argument. Assuming that the program has previously assigned values to the invocation arguments **input**, **y**, **ydot**, **w**, **r**, the submodel invocation

```
invoke quad(input, y, ydot, w, r)
```

generates compiled in-line code equivalent to

```
d/dt y = ydot
d/dt ydot = input - w * y - r * ydot
```

Submodel invocation arguments must be names of previously defined scalars, vectors, or matrices, not expressions as for the user-defined functions in Section 3-16. An error message gives a warning if declaration and invocation arguments do not match. A submodel can be invoked as often as needed, with the same or different invocation arguments.

Submodel definitions may contain any legal DYNAMIC-segment assignments and differential equations, including vector/matrix operations. Submodel definitions can use scalar and array variables as dummy arguments and may also involve additional variables and parameters common to all invocations. As in the case of user-defined functions (Section 3-16), it is convenient to label dummy arguments such as **x\$** with a dollar sign, but this is not necessary. After a dummy-argument name is used in a submodel declaration, it can no longer be used elsewhere; it is “protected” by an error message to prevent side effects. Program displays and listings automatically indent definition lines, as shown in our example.

The experiment protocol must declare arrays for subscripted variables, vectors, or matrices used as invocation arguments in DYNAMIC program segments; note that different array dimensions can be used for different invocations. Arrays used as dummy arguments in **SUBMODEL** declarations must also be declared. Since such dummy arrays are never filled with actual values, memory can be saved by setting all dummy-array dimensions to 1.

In the submodel defined by

```
SUBMODEL normalize(v$, v1$)
  DOT vnormsq = v$ * v$ | vnn = 1/sqrt(xnormsq)
  Vector v1$ = vnn * v$
end
```

the scalars **vnormsq** and **vnn** are “global” parameters used as intermediate results in all instances of the submodel. When one wants to invoke **normalize** to obtain normalized versions **U** and **V** of two different vectors **u** and **v** by programming

```
invoke normalize(u, U) | invoke normalize(v, V)
```

the dummy arrays **v\$, v1\$** and the invoked arrays **u, U, v, V**, must first be declared, say, with

```
ARRAY v$[1], v1$[1] | ARRAY u[m], U[m], v[n], V[n]
```

Submodel definitions can contain user-defined functions and may invoke other submodels (nested submodels). But nested and recursive submodel definitions, and also recursive submodel invocations, are illegal.

(b) Submodels with Differential Equations

For submodels involving differential equations (**d/dt** or **Vectr d/dt** statements) all invoked differential-equation state variables must be declared with **STATE** declarations in the experiment protocol (Section 3-1). This is necessary even for scalar state variables, not just for state-variable arrays. But dummy state variables do not need **STATE** declarations for they are never used in actual differential equations. For example, the definition and invocation of the mass–spring submodel **quad(x\$, y\$, ydot\$, a\$, b\$)** in Section 3-17a requires the declaration

```
STATE y, ydot
```

but the dummy variables **y\$** and **ydot\$** need not be declared as state variables.

3-18. Dealing with Sampled-data Assignments, Limiters, and Switches

A user-defined function involving sampled-data assignments, limiters and/or switches (Table 3-1) generates only one line of DYNAMIC-segment code, and can be thus programmed following an **OUT**, **SAMPLE m**, or **step** statement as

discussed in Sections 2-10 and 2-11. However, submodels can generate multiple lines that cannot be separated by **OUT**, **SAMPLE m**, or **step** statements in a submodel definition. As a result, a submodel must generate only differential-equation-system (“analog”) code, only limiter/switch operations operating on analog variables, or only sampled-data operations. Sampled-data assignments can safely include limiter/switch operations.

It is then, strictly speaking, incorrect to invoke the submodel defined by

```
SUBMODEL signal(y$, p$, w$)
  d/dt y$ = w$ * p$
  p$ = sgn(p$ - y$)
end
```

to produce triangle waves and square waves in the manner of Section 2-17. Serendipitously, the resulting code usually works anyway, presumably because we are only integrating a constant input equal to either **a** or **-a**.

REFERENCES

1. D. Hetrick, *Dynamics of Nuclear Reactors*, University of Chicago Press, Chicago, 1971.
2. G. A. Korn, A Simulation-model compiler for all seasons, *Simulation Practice and Theory*, **9**, 2001, pp. 21–25.
3. G. A. Korn, *Interactive Dynamic System Simulation with Microsoft Windows*, Taylor and Francis, London, 1998.
4. G. F. Franklin, *Digital Control of Dynamic Systems*, Addison-Wesley, Reading, MA, 1990.
5. G. A. Korn, A new software technique for submodel invocation, *Simulation*, March 1987, pp. 93–97.

4

Parameter-influence Studies, Model Replication, and Monte Carlo Simulation

PARAMETER-INFLUENCE STUDIES AND VECTORIZATION

4-1. Exploring the Effects of Parameter Changes

Parameter-influence studies explore effects of different combinations of model and experiment parameters. Initial state-variable values are treated simply as extra model parameters. For a system of differential equations or difference equations, for example,

$$(\mathbf{d}/\mathbf{dt}) \mathbf{x} = \mathbf{f}(\mathbf{t}; \mathbf{x}, \mathbf{y}; \mathbf{a}, \mathbf{b}, \dots) \quad \mathbf{y} = \mathbf{g}(\mathbf{t}; \mathbf{x}; \mathbf{c}, \mathbf{d}, \dots) \quad (4-1)$$

with suitably differentiable functions \mathbf{f} and \mathbf{g} , we can measure the sensitivity of $\mathbf{x} = \mathbf{x}(\mathbf{t})$ and $\mathbf{y} = \mathbf{y}(\mathbf{t})$ to small changes in the parameter \mathbf{a} by computing time histories of the parameter-sensitivity coefficients $\mathbf{u}(\mathbf{t}) \equiv \partial \mathbf{x} / \partial \mathbf{a}$ and $\mathbf{v}(\mathbf{t}) \equiv \partial \mathbf{y} / \partial \mathbf{a}$. Differentiation of the system equations (4-1) with respect to the parameter \mathbf{a} produces the differential-equation system

$$(\mathbf{d}/\mathbf{dt})\mathbf{u} = (\partial \mathbf{f} / \partial \mathbf{x})\mathbf{u} + (\partial \mathbf{f} / \partial \mathbf{y})\mathbf{v} + \partial \mathbf{f} / \partial \mathbf{a} \quad \partial \mathbf{y} / \partial \mathbf{a} = (\partial \mathbf{g} / \partial \mathbf{x})\mathbf{u} \quad (4-2)$$

In principle, the parameter-sensitivity equations (4-2) can be solved together with the given system equations (4-1) to produce time histories of \mathbf{u} and \mathbf{v} .

Parameter-influence coefficients are theoretically interesting. But for a system with \mathbf{N} equations [Eq. (4-1)], in general, $2\mathbf{N}$ equations (4-1) and (4-2) have to be solved even when only the sensitivity of one system variable to a single parameter is needed. Even that reveals only effects of small parameter changes. It is usually easier to just solve the given system equations for different parameter combinations (Sections 4-2 and 4-3).

Monte Carlo simulation with randomly perturbed parameter values (Section 4-4) is also a form of parameter-influence study and permits, for instance, statistical regression of performance measures on parameter values [1].

4-2. Repeated Runs and Model Replication (Vectorization)

(a) A Simple Repeated-run Study

Repeated-run parameter-influence studies simply repeat simulation runs with different parameter values. As an example, the response $\mathbf{x}(\mathbf{t})$ of a damped harmonic oscillator after an initial displacement $\mathbf{x}(0) = 1$ is modeled by the DYNAMIC program segment

```
DYNAMIC
d/dt x = xdot
d/dt xdot = - ww * x - r * xdot
X = x - scale | -- offset the display
dispt X | -- display
```

(4-3)

We let $\mathbf{xdot}(0)$ default to 0. A small repeated-run parameter-influence study explores the effects of different positive damping coefficients \mathbf{r} with the experiment-protocol script

```
TMAX = 0.5 | DT = 0.0001 | NN = 1001
ww = 400 | -- fixed system parameter
x = 1 | -- given initial displacement
n = 5 | -- number of simulation runs
for i = 1 to n | -- set parameter values
  r = 5 * i
  drunr
next
```

(4-4)

This experiment protocol calls $n = 5$ simulation runs of the model (4-3) with the damping coefficient r successively set to 5, 10, 15, 20, and 25 (Figure 4-1).

(b) Model Replication

Instead of repeating simulation runs with different parameter values, a single simulation run can exercise $n = 5$ replicas of the model with different parameter values. We replicate the model (4-3) by declaring the state variables \mathbf{x} , $\mathbf{\dot{x}}$ and the parameter r as n -dimensional vectors

$$\mathbf{x} \equiv (x[1], x[2], \dots, x[n]) \quad \mathbf{\dot{x}} \equiv (\dot{x}[1], \dot{x}[2], \dots, \dot{x}[n])$$

$$\mathbf{r} \equiv (r[1], r[2], \dots, r[n])$$

This is done with the new experiment-protocol script

```

TMAX=0.5 | DT=0.0001 | NN=1000
ww = 400 | -- fixed system parameter
-----
n = 5 | STATE x[n], xdot[n] | ARRAY r[n]
-----
for i = 1 to n
    x[i] = 1 | -- set n initial displacements
    r[i] = 5 * i | -- set n parameter values
next
drun
    
```

(4-5)

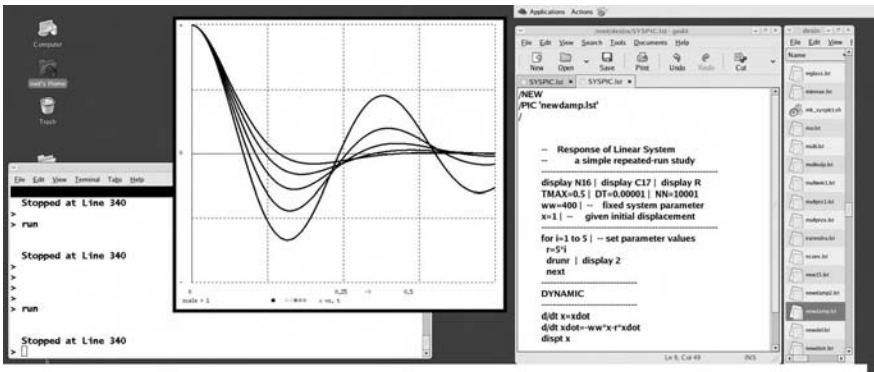


FIGURE 4-1. Linux dual-screen display of the small repeated-run study in Section 4-2a. Double-clicking the file **newdamp.lst** in the file-manager window on the right has loaded the program into DESIRE. The original display was in color.

This script “fills” the parameter array **r** with the **n** desired parameter values¹

$$\mathbf{r}[1] = 5, \mathbf{r}[2] = 10, \mathbf{r}[3] = 15, \mathbf{r}[4] = 20, \mathbf{r}[5] = 25 \quad (4-6)$$

Next, a new DYNAMIC program segment replaces the model (4-3) with the corresponding vectorized model

DYNAMIC

Vectr d/dt **x** = **xdot**

Vectr d/dt **xdot** = - **ww** * **x** - **r** * **xdot**

dispt **x[1], x[2], x[3], x[4], x[5]** | -- display 5 curves (4-7)

DESIRE’s vectorizing compiler (Sections 3-2 and 3-3) automatically compiles this vector model into **n** replicated scalar state-equation systems

$$\begin{aligned} \mathbf{d/dt\ x[i]} &= \mathbf{xdot[i]} \\ \mathbf{d/dt\ xdot[i]} &= - \mathbf{ww} * \mathbf{x[i]} - \mathbf{r[i]} * \mathbf{xdot[i]} \quad (\mathbf{i} = 1, 2, \dots, \mathbf{n}) \end{aligned} \quad (4-8)$$

The **n** state-variable initial values **xdot[i]** default to 0, and **ww** is a scalar parameter common to all **n** models. Our program effectively replicates the original model (4-3) **n** times with different parameter values [Eq. (4-6)] and exercises all **n** replicated models in a single simulation run. The resulting solutions **x[1], x[2], x[3], x[4], x[5]** are exactly the same as the solutions **x** obtained for **r = 5, 10, 15, 20, 25** in Figure 4-1.

The DESIRE vector compiler makes model replication automatic. Vectorization works for both differential-equation systems and sampled-data assignments. Replicated models can involve user-defined functions, table-lookup functions, and submodels; each function or submodel definition is necessarily the same for all **n** models. Vector and matrix operations (Chapter 3), and also time delays and **store/get** operations (see the DESIRE reference manual in CD) cannot be replicated.

Model replication improves computing speed by eliminating the runtime loop and run-starting overhead of repeated-run studies. Model replication requires extra memory; a compact vector model can generate a large equation system. DESIRE currently admits up to 150,000 double-precision defined variables, plus up to 40,000 differential-equation state variables for fixed- and

¹ Instead of filling state and parameter arrays with a program loop, array elements can also be given individual assignments such as **b[17] = 9.8887**, and multiple vector arrays can be filled with a single **data/read** assignment such as **data 0.1, -7.8, cos (gamma), 12.1, ... | read b,x**

variable-step Runge–Kutta integration rules. This is enough for 1000 40th-order differential-equation models. Our variable-step/variable-order Gear-type and Adams integration rules need more memory and are therefore limited to 600 state variables. For larger problems, there is an easy combination technique: we simply program repeated runs of a vectorized model with new parameter values (Section 4-11).

(c) Dealing with Multiple Parameters

Our example varied only one parameter, but we can readily deal with multiple parameters. Suppose we have a variable parameter **a** with **n1 = 4** values

– 5.0, – 2.0, 3.0, 4.0

and a second variable parameter **b** with **n2 = 3** values,

2.7, 0, 10.0

Our experiment-protocol script must then assign the **n = n1 * n2 = 12** values

– 5.0, – 2.0, 3.0, 4.0; – 5.0, – 2.0, 3.0, 4.0; – 5.0, – 2.0, 3.0, 4.0

to **a[1], a[2], ..., a[n]**, and also the corresponding **n** values

2.7, 2.7, 2.7, 2.7; 0, 0, 0, 0; 10.0, 10.0, 10.0, 10.0

to **b[1], b[2], ..., b[n]**. To simplify this task, we declare and fill an **n1**-dimensional array **aa** and an **n2**-dimensional array **bb** with

```
n1 = 4      |      n2 = 3      |      ARRAY aa[n1], bb[n2]  
data – 5.0, – 2.0, 3.0, 4.0; 2.7, 0, 10.0      |      read aa, bb
```

We now declare and fill the desired **n**-dimensional parameter arrays **a** and **b** with

```
n = n1 * n2 |      ARRAY a[n], b[n]  
for k = 1 to n2  
  for i = 1 to n1  
    a[i + (k - 1) * n1] = aa[i]      |      b[i + (k - 1) * n1] = bb[k]  
  next  
next
```

The **[i + (k - 1) * n1]** th model has the parameter combination **aa[i], bb[k]**. The smaller arrays **aa** and **bb** are much easier to read, write, and store than

the repetitious **a** and **b** arrays. This procedure can be extended to three or more parameters.

4-3. Programming Parameter-influence Studies

(a) Introduction

Before vectorizing a model, we usually check it out in scalar form. This also simplifies sorting defined-variable assignments (Sections 1-9 and 2-1). When the simulation works, one will want to program output procedures specifically designed to evaluate the effects of changing parameters.

Our toy example was simple enough. But real parameter-influence studies involve multiple parameters and possibly very many parameter combinations. We must vary

- the design parameters that we want to optimize under different conditions,
- parameters that represent these different conditions (e.g., different temperatures and different initial conditions).

As we noted in Section 1-17, simulations quickly produce large volumes of time-history graphs and numerical tables. Meaningful evaluation of such results is a very real problem.

DESIRE experiment-protocol commands can list successive parameter settings and simulation results—even entire arrays—in a journal file.² This preserves the data, but may not relate successive results in a meaningful way. A better plan is to let experiment-protocol scripts or DYNAMIC program segments write data tables directly into *space-delimited text files*. These can then be fed to standard spreadsheet and relational-database programs for analysis (data mining), presentations, and storage. DESIRE experiment-protocol scripts can readily access external programs and exchange file data with them.

(b) Measures of System Effectiveness

A *system* combines hardware, people, and/or modes of operation for some purpose. Then, the very definition of an engineering system requires that quantitative measures of its effectiveness be defined. These measures are normally numerical functions of system parameters. We often use cost-related functionals such as integrals of system-variable time histories, such as the control-system error measures in Section 1-14. Parameter-influence studies must define effectiveness measures and compute their values for each parameter combination.

² Refer to the DESIRE reference manual in the book CD.

We shall want to maximize measures of system effectiveness (or minimize cost measures) as functions of system parameters. More often than not, however, practical design is not the result of straightforward global optimization but involves compromises:

- Conflicting measures (say cost and performance) may need individual consideration—a single measure (e.g., performance per unit cost) may not do.
- One may have to compromise between performance results obtained under different conditions (e.g., different signal amplitudes, temperatures, or initial conditions).

Simulation results are only raw material for making such decisions. The user has to make intelligent compromises.

(c) Crossplotting Results

Consider a model producing a performance measure such as the control-system integrated squared error **ISE** in Section 1-14,

$$d/dt \text{ ISE} = (x - u)^2 \quad (\text{ISE, integral squared error}) \quad (4-9)$$

Its value **ISE** = **ISE(t0 + TMAX)** at the end of a simulation run can be a useful control-system performance measure. To see clearly how **ISE** depends on a system parameter, say the servo damping coefficient **r**, we may want to plot a graph of **ISE** versus **r**.

1. An **n**-run repeated-run study (Section 4-2a) makes **n** simulation runs with **n** parameter values of **r** = **r0**, **r0 + DELr**, **r0 + 2 DELr**, ... and produces a corresponding **ISE** value [Eq. (4-9)] at the end of each run. If a runtime time-history display is not needed, the experiment-protocol script can crossplot **ISE** versus **r** as successive runs proceed:

```
for i = 1 to n
  r = r + (i - 1) * DELr
  drun
  plot r, ISE, c | -- (c = 1, 2, ... is a graph color)
  reset
next
```

Instead, one may want to watch a runtime time-history display and save corresponding values of **r** and **ISE** in two **n**-dimensional arrays declared with

```
ARRAY rr[n], ise[n]
```

The **plot** line in the above script loop can be simply replaced with

```
rr[i] = r      |      ise[i] = ISE
```

One can then plot, cross-list, and analyze corresponding **ise[i]** and **r[i]** values later on.

2. A replicated-model (vectorized) study (Section 4-2b) inherently implies declaration of an **n**-dimensional parameter array (vector) **r** and an **n**-dimensional state vector **ISE**. The experiment-protocol script fills the **r** array and makes a single simulation run:

```
ARRAY r[n], ... | STATE x[n], ... , ISE[n]  
.....  
for i = 1 to n | --      set parameter values  
  r = r0 + (i - 1) * DELr  
  next  
-----  
drun
```

The corresponding **n**-dimensional arrays (vectors) **r** and **ISE** are then available for crossplotting, or for any other purpose.

(d) Maximum/Minimum Selection

If there is an **n**-dimensional array (vector) of performance-measure values, say the array **ISE** in Section 4-3c, the maximum-selection techniques of Section 3-8 can conveniently determine the index **i = I** of the largest or smallest performance-measure value **ISE[i]** and compute that value. One should remember, though, that maximum/minimum selection works only in DYNAMIC program segments and is therefore more easily applied in vectorized parameter-influence studies than in repeated-run studies.

(e) Iterative Parameter Optimization

Parameter-influence studies produce performance measures as functions **F(a, b, ...)** of parameter values **a, b, ...**. Repeated-run simulation studies can relate a selection of successive parameter combinations **a, b, ...** to past results in such a way that **F(a, b, ...)** converges to its global maximum or minimum.

A primitive experiment protocol for minimizing **F** might simply change one parameter at a time in the direction that turns out to decrease **F**. This may work if **F(a, b, ...)** is continuous and has only one minimum and no “flat spots” where it is locally constant. As an example, the following experiment-protocol script finds the value of a servomechanism damping coefficient **r** that minimizes the integral squared error **ISE** (see also Section 1-14).

(set TMAX, DT, *parameters*, and *initial conditions* ...)

.....

```

drun | -- initial run with trial value r produces ISE
repeat
  oldISE = ISE
  r = r + DELr | -- increment r
  reset | drun | -- run with r + DELr
  DELISE = ISE - oldISE | -- measure the gradient
  if abs(DELISE) < crit then exit | -- no more change
  else proceed
  r = r - DELr - opgain * DELISE | -- working step
  reset | drun
  until 0 > 1 | -- keep trying
write 'optimal values: r = ' ; r, 'ISE = ' ; ISE | -- report result

```

Such simple one-parameter optimizations make nice demonstrations [2], but real-life optimization studies are usually far more difficult. They must handle multiple parameters and performance-measure “landscapes” with flat spots and local minima. Every parameter-improvement step will require multiple, cleverly designed trial evaluations of $\mathbf{F}(\mathbf{a}, \mathbf{b}, \dots)$. Parameter optimization is a complicated subject requiring separate study [3]. For serious optimization projects, the simulation experiment protocol may have to interact with an external specialized optimization program [3].

RANDOM PROCESSES AND RANDOM PARAMETERS

4-4. Random Processes and Monte Carlo Simulation

A random process generates sample functions $\mathbf{x}(\mathbf{t})$ that depend on random parameters, random initial conditions, and/or random inputs. By “random” we mean quantities whose behavior can be predicted only by the values of statistics. Statistics are functions of repeated measurements, for example, sample averages and statistical relative frequencies. We use statistics because serendipitous experience indicates that suitably defined statistics often fluctuate less than individual measurements and are thus more predictable. Statistics, in fact, often fluctuate less and less as the sample size increases.³

³ This fortunate fact—an *empirical law of large numbers*—is not derived from probability theory but is based on observations, just like a law of nature in physics. Similar *mathematical* laws of large numbers (e.g., the central limit theorem) relate to expected values and probabilities, that is, to properties of models. Validation of mathematical laws by empirical laws of large numbers indicates that specific probability models can match and predict real-world experience.

Probability models describe random processes in terms of joint probability distributions of different sample values $\mathbf{x}(\mathbf{t1}), \mathbf{x}(\mathbf{t2}), \dots$ [4,5]. Such models try to fit observed statistics with theoretical concepts such as probabilities and expected values. A random process is stationary if its joint probability distributions are unaffected when we shift the time origin by adding the same time shift τ to all sampling times \mathbf{ti} .

We are going to simulate random processes generated by dynamic systems with *random parameters* (this includes random initial conditions) and/or *random time-function inputs (noise)*. A Monte Carlo simulation study repeats or replicates simulations of such systems to produce a sample of different random-process sample functions $\mathbf{x}(\mathbf{t})$. We then compute statistics such as sample averages over corresponding values read from different sample functions. Monte Carlo statistics typically measure various aspects of system performance.

4-5. Generating Random Parameters and Random Initial Values

Experiment-protocol scripts generate random parameter values $\mathbf{a}, \mathbf{b}, \dots$ with assignments such as

$\mathbf{a} = \mathbf{ran}()$ | $\mathbf{b} = \mathbf{cos}(\mathbf{ran}() + \mathbf{c})$ | \dots

State-variable initial values are simply additional parameters. Each call of the DESIRE library function **ran()** produces a new sample of a pseudorandom-noise sequence. Pseudorandom noise is really not random but a programmed number sequence that repeats after a large number of samples. **ran()** output is uniformly distributed between -1 and 1 with theoretical mean 0 and variance $1/3$. Different samples are uncorrelated but not statistically independent; this problem will be discussed in Section 5-4. The experiment-protocol command **seed m** can start or restart the noise sequence with a specific fixed value. This can be useful for testing programs.

Various functions of the uniformly distributed **ran()** output can produce samples with different known probability distributions (see the references to Chapter 5). Sums $\mathbf{y} = \mathbf{ran}() + \mathbf{ran}() + \dots$ with $4-7$ terms are approximately Gaussian with mean 0 and variance $\mathbf{N}/3$, where \mathbf{N} is the number of terms. But multiple samples of \mathbf{y} are not necessarily jointly Gaussian.

MONTE CARLO SIMULATION OF DYNAMIC SYSTEMS

4-6. Repeated-run Monte Carlo Simulation

(a) Taking Statistics on Repeated Simulation Runs

Repeated-run Monte Carlo simulation programs loop to exercise a DYNAMIC program segment \mathbf{n} times with new random inputs and then take statistics on the results.

In the following experiment-protocol script, each of n passes through a program loop assigns new random values to a parameter b and to a state-variable initial value $q(0)$ and then calls a simulation run. The n successive simulation runs produce end-of-run sample values $x[i] = x(t_0 + TMAX)$ of a system time history $x(t)$ for $i = 1, 2, \dots, n$. x can be a state variable or a defined variable; it is usually a system-performance measure similar to **ISE** in Section 4-3. After completing n runs, the program computes

- the sample average $xAvg = \langle x \rangle = (x[1] + x[2] + \dots + x[n])/n$
- the sample mean square $xxAvg = \langle x^2 \rangle = (x^2[1] + x^2[2] + \dots + x^2[n])/n$
- the sample variance $xVar = xxAvg - xAvg^2$.

We employ the convenient **DOT** operations described in Section 3-7 to produce the n -term sums $xSum$ and $xxSum$ needed to compute $xAvg$ and $xxAvg$.

(first set fixed system parameters, initial conditions, etc.)

```

n = 1000 | ARRAY x[n] | --      declare an array of sample values
--
for i = 1 to n | --                      Monte Carlo loop
  b = b0 + B * f1(ran()) | --      set a new random parameter value
  q = q0 + C * f2(ran()) | --      set a new random initial value
  --
  drunr | --      make a simulation run and reset state variables
  x[i] = X | --    read successive sample values of x =X(t0 + TMAX)
next
--
                                now compute statistics
DOT xSum = X * 1 | xAvg = xSum/n
DOT xxSum = x * x | xxAvg = xxSum/n
  xVar = xxAvg - xAvg^2 | s = sqrt(xVar)

```

drunr (or **drun** | **reset**) calls a simulation run and then resets differential-equation-system initial values.⁴

It would be just as easy to take statistics on two or more performance measures x, y, \dots . More complicated statistics such as correlation and regression coefficients, probability and probability-density estimate, and test statistics such as t and χ^2 can all be computed as functions of various sample averages (see also Section 4-9). As always, Monte Carlo results have to be checked with different pseudorandom-noise generators (see also Section 5-11).

⁴ For simplicity, we have assumed that the only state variables are differential-equation state variables (see also Section 2-5).

(b) Sequential Monte Carlo Studies

Instead of computing Monte Carlo statistics after n repeated simulation runs, we can accumulate sample averages after every simulation run. The following experiment-protocol script first initializes the sample averages \mathbf{xAvg} and \mathbf{xxAvg} and then again loops to make n simulation runs with new parameter and initial-condition values. At the end of the i th run, the program reads $\mathbf{x} = \mathbf{x}(t_0 + TMAX) = \mathbf{x}(i)$ and updates the statistics values:

```

xAvg = 0 | xxAvg = 0 | --           initialize statistics computation
for i = 1 to n | --                 Monte Carlo loop
    b = b0 + B * f1(ran()) | --       set a new random parameter value
    q = q0 + C * f2(ran()) | --       set a new random initial value
    --
    drunr | --                       make a simulation run and reset state variables
    x[i] = X | --                     read successive sample values of  $\mathbf{x} = \mathbf{X}(t_0 + TMAX)$ 
    ----- now accumulate statistics!
    xAvg = xAvg + (x - xAvg)/n
    xxAvg = xxAvg + (x^2 - xxAvg)/n
    xVar = xxAvg - xAvg^2
next | --                           and loop back

```

This technique can save time, for it permits us to terminate the study when the sample variance has become sufficiently small (sequential Monte Carlo simulation).

(c) Example: Effects of Gun-elevation Errors on the 1776 Cannon

We will study a continuous random process generated by a differential-equation system with a random parameter, specifically a random initial value. Similar programs apply directly to many Monte Carlo studies of manufacturing-tolerance effects.

Simulation of the 1776 cannon⁵ in Figure 4-2 has been used as a textbook problem for over 50 years [6,7]. Assuming that the wind force $\mathbf{W}(t)$ is zero, the only forces acting on the spherical cannonball are its weight \mathbf{mg} and aerodynamic drag opposing the velocity vector. Airspeed is relatively low, so that the drag is roughly proportional to the square of the velocity. Referring to Figure 4-2, the equations of motion in the horizontal and vertical directions are

$$\begin{aligned}
 (d/dt) \mathbf{x} &= \mathbf{xdot} & (d/dt) \mathbf{xdot} &= -R v^2 \cos \theta = -R v \mathbf{xdot} \\
 (d/dt) \mathbf{y} &= \mathbf{ydot} & (d/dt) \mathbf{ydot} &= -R v^2 \sin \theta - g = -R v \mathbf{ydot} - g
 \end{aligned}$$

⁵1776 gun elevations were not really affected by manufacturing errors. Elevations of land-based guns were usually set with wedges under the rear part of the barrel, and naval-gun elevation also required judgment of the ship's roll angle. Either way, there were lots of random errors.

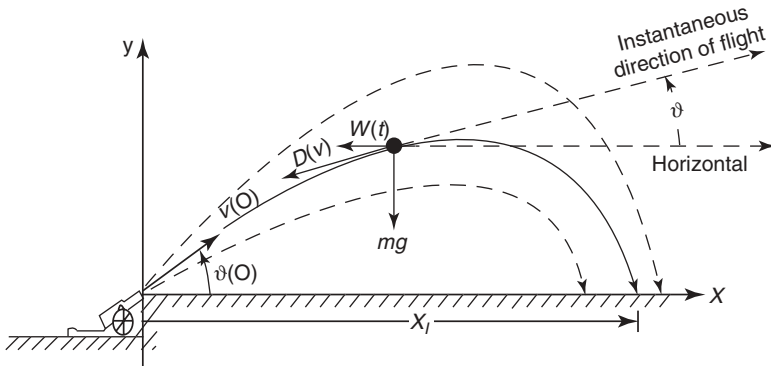


FIGURE 4-2. Cannon geometry (based on Reference [6]). We assume that the wind force $\mathbf{W}(t)$ is zero.

with

$$v = \sqrt{\dot{x}^2 + \dot{y}^2}$$

The acceleration due to gravity $\mathbf{g} = 32.2 \text{ ft/s}^2$ and $\mathbf{R} = 7.5\text{E-}05 \text{ ft}^{-1}$ is the drag coefficient divided by the projectile mass. The trajectory of each shot is then determined by the initial muzzle position $\mathbf{x}(0) = \mathbf{y}(0) = \mathbf{0}$ and the initial velocity components

$$\dot{x}(0) = v_0 * \cos(\theta_0) \quad \dot{y}(0) = v_0 * \sin(\theta_0) \quad (4-10)$$

θ_0 is the gun elevation angle, and $v_0 = 900 \text{ ft/s}$ is the given muzzle velocity.

Assuming level ground, the impact abscissa \mathbf{x}_l is the value of \mathbf{x} where $\mathbf{y} = 0$ at the end of a trajectory. A good way to read \mathbf{x}_l is with the track-hold difference equation

$$\mathbf{x}_l = \mathbf{x}_l + \text{switch}(\mathbf{y}) * (\mathbf{x} - \mathbf{x}_l)$$

(Section 2-16b), which causes \mathbf{x}_l to track \mathbf{x} while and then holds the \mathbf{x} value. The initial value of the difference-equation state variable \mathbf{x}_l defaults to 0. To aim the cannon, we set the elevation angle θ_0 to obtain a desired impact abscissa \mathbf{x}_l , say $\theta_0 = 70 * \text{PI}/180$. Our Monte Carlo study then adds random perturbations to this nominal gun elevation and determines their effect on the sample average and sample variance of the impact coordinate \mathbf{x}_l . To get approximately Gaussian-distributed elevation errors, we set

$$\theta_0 = 70 * \text{PI}/180 + a * (\text{ran}() + \text{ran}() + \text{ran}() + \text{ran}())$$

Since **ran()** is uniformly distributed between -1 and 1 with expected value 0 and theoretical variance $1/3$, we have

$$E\{\theta_0\} = 70 \text{ PI}/180$$

$$\text{Var}\{\theta_0\} = 4 * a^2/3$$

Figure 4-3 shows time histories of **x(t)** and the track-hold output **xl(t)** for a few simulation runs together with the complete program for the repeated-run Monte Carlo study. The program also displays the resulting sample average **xavg** and the sample statistical dispersion **s = sqrt(abs(xxavg - xavg^2))** of the impact abscissa **xl** after **n** runs.

4-7. Vectorized (Model-replicating) Monte Carlo Simulation

(a) Vectorized Monte Carlo Study of the 1776 Cannon Shot

Model-replicating or vectorized Monte Carlo simulation was originally developed for supercomputer studies of small physics models, where the repeated-run program overhead is especially significant. As we saw in Section 4-2b, our vector compiler conveniently implements vectorization on inexpensive personal computers, where its advantages—simpler programs and high speed, at least for small models—are welcome indeed. As an added bonus, vectorization can also help check the quality of pseudorandom noise (Section 5-10).

Instead of repeating simulated cannon shots as in Section 4-6, the experiment protocol in Figure 4-4 declares **n**-dimensional state-variable and arrays (vectors)

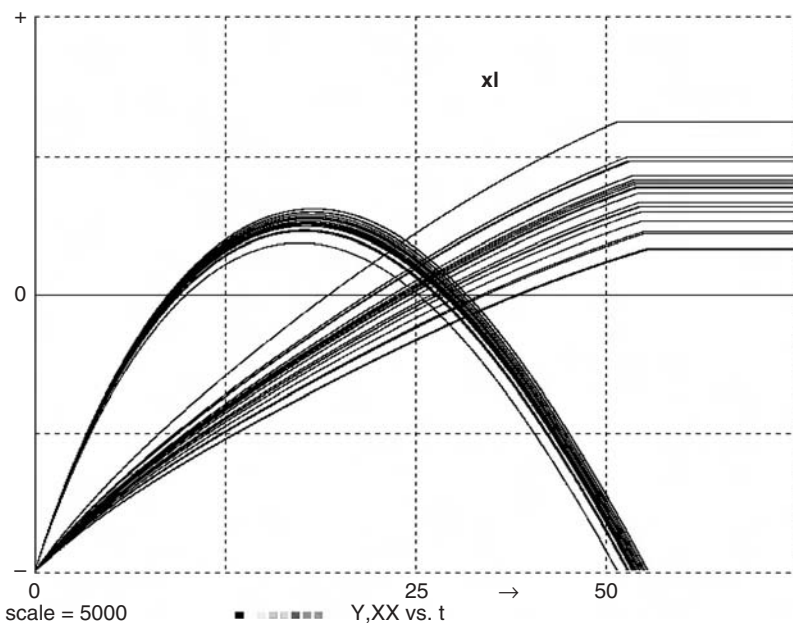
STATE x[n], y[n], xdot[n], ydot[n] | ARRAY theta0[n], v[n], xlimpact[n]

and loops to “fill” each of the arrays **theta0**, **xdot**, and **ydot** with **n** different random initial values:

```
for i = 1 to n | -- noisy elevation angle in radians
  theta0[i] = 70 * PI/180 + a * (ran()+ran()+ran()+ran())
  xdot[i] = v0 * cos(theta0[i]) | ydot[i] = v0 * sin(theta0[i])
next
```

A vectorized DYNAMIC program segment then effectively replicates the cannonball model and the output track/hold operation **n** times:

```
Vector v = sqrt(xdot^2 + ydot^2) | -- a defined variable
Vectr d/dt x = xdot | Vectr d/dt y = ydot | -- equations of motion
Vectr d/dt xdot = - R * v * xdot | Vectr d/dt ydot = - R * v * ydot - g
--
step
Vector xlimpact = xlimpact + swtch(y) * (x - xlimpact) | -- track-hold
```

— REPEATED-RUN MONTE CARLO: 1776 CANNON

DT = 0.008 | TMAX = 50 | NN = 5000 | scale=5000

R = 7.5E-05 | g = 32.2

v0 = 900 | —

muzzle velocity
noise amplitude

a = 0.03 | —

n = 1000 | ARRAY xImpact[n] | —

sample values

for i = 1 to n | —

elevation in radians
initialize track-hold

xl = 0 | —

theta0 = 70 * PI/180 + a * (ran()+ran()+ran()+ran())

xdot = v0 * cos(theta0) | ydot = v0 * sin(theta0)

drunr | display 2 | — run, don't erase display

xImpact[i] = xl | — read the impact abscissa xl

next

COMPUTE STATISTICS AFTER n RUNS

DOT xSum = xImpact * 1 | xAvg = xSum/n

DOT xxSum = xImpact * xImpact | xxAvg = xxSum/n

s = sqrt(abs(xxAvg - xAvg^2)) | — dispersion

write "xAvg = ",xAvg," s = ",s

DYNAMIC

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

d/dt x = xdot | d/dt y = ydot

d/dt xdot = - R * v * xdot | d/dt ydot = - R * v * ydot-g

step

xl = xl + swtch(y) * (x - xl) | — hold the impact abscissa

```

-- VECTORIZED MONTE CARLO STUDY: 1776 CANNON
-----
DT = 0.008 | TMAX = 50 | NN = 5000 | scale=5000
R = 7.5E-05 | g = 32.2
v0 = 900 | -- muzzle velocity
a = 0.03 | -- noise amplitude
--
n = 1000 | STATE x[n], y[n], xdot[n], ydot[n]
ARRAY theta0[n], v[n], xImpact[n]
--
for i= 1 to n | -- noisy elevation angle in radians
  theta0[i] = 70 * PI/180 + a * (ran()+ran()+ran()+ran())
  xdot[i] = v0 * cos(theta0[i]) | ydot[i] = v0 * sin(theta0[i])
  next
--
drun | make a single simulation run ...
      ... and then compute statistics
--
DOT xSum = xImpact * 1 | xAvg = xSum/n
DOT xxSum = xImpact * xImpact | xxAvg = xxSum/n
s = sqrt(xxAvg - xAvg^2)
write "xAvg = ";xAvg," s = ";s
-----
DYNAMIC
-----
Vector v = sqrt(xdot^2 + ydot^2)
Vectr d/dt x = xdot | Vectr d/dt y = ydot
Vectr d/dt xdot = - R * v * xdot | Vectr d/dt ydot = - R * v * ydot - g
--
step
Vector xImpact = xImpact + switch(y) * (x - xImpact) | -- n track-holds

```

FIGURE 4-4. Complete program for a vectorized Monte Carlo study of the 1776 cannonball problem. All initial **xImpact[i]** default to 0.

Figure 4-4 shows the complete program. The vectorized cannonball study produced essentially the same results as the repeated-run study, as expected.

One thousand repeated runs and the statistics computation took 5 s on a 2.4-GHz Athlon64, and the equivalent vectorized study took 3.4 s. This speed advantage is typical for small models. The repeated-run overhead saved by vectorization becomes less significant as the model size increases.

FIGURE 4-3. This repeated-run Monte Carlo study determines the impact-range dispersion of a 1776 cannon shot due to random elevation-setting errors. A track-hold difference equation (Section 2-16b) holds the impact coordinate.

(b) Interactive Monte Carlo Simulation: Computing Time Histories of Statistics with Compiled DOT Operations

Vectorized Monte Carlo simulation has another interesting and important feature. Since a single simulation run samples all n replicated models at each point of time, one can compute and display time histories of statistics, and observe the results of parameter changes as the simulation run proceeds. Such interactive Monte Carlo simulation was formerly possible only with very fast (and very inaccurate) analog computers [7].

Runtime statistics computations are needed only at output-sampling times, not at every derivative call. We will thus save time by programming DYNAMIC-segment statistics computations following an **OUT** or **SAMPLE m** statement (Section 1-6). As noted in Section 4-6a, most statistics are functions of sample averages. A vectorized DYNAMIC program segment computes the sample averages

$$\begin{aligned} \text{qAvg}(t) &= (\text{q}[1] + \text{q}[2] + \dots + \text{q}[n])/n \\ \text{qqAvg}(t) &= (\text{q}^2[1] + \text{q}^2[2] + \dots + \text{q}^2[n])/n \end{aligned}$$

of a replicated system variable $\mathbf{q} = \mathbf{q}(t)$ at each sampling time t with

OUT
DOT $\text{qSum} = \mathbf{q} * 1 \mid \text{qAvg} = \text{qSum}/n$
DOT $\text{qqSum} = \mathbf{q} * \mathbf{q} \mid \text{qqavg} = \text{qqSum}/n$

The value of $1/n$ can be precomputed by the experiment protocol to avoid time-consuming divisions by n . Unlike in Section 4-6, we are using compiled **DOT** operations that, like DESIRE vector assignments, involve no program-loop overhead (Section 3-7).

We could add runtime computation of $\mathbf{xAvg}(t)$ and $\mathbf{yAvg}(t)$ to the vectorized cannonball study in Figure 4-4 and display the average trajectory (graph of $\mathbf{yAvg}(t)$ versus $\mathbf{xAvg}(t)$). We shall exhibit more interesting applications in Sections 5-8 and 5-9.

4-8. Statistical Relative Frequencies, Sample Ranges, and Other Statistics

For any random variable \mathbf{x} in our Monte Carlo model, the sample statistical relative frequency \mathbf{hh} of an event $\{\mathbf{a} < \mathbf{x} < \mathbf{b}\}$ ($\mathbf{a} < \mathbf{b}$) is the fraction of our n process samples where $\mathbf{a} < \mathbf{x} < \mathbf{b}$ is true. \mathbf{hh} estimates the probability of the event. Instead of counting events, we measure \mathbf{hh} as the sample average \mathbf{uAvg} of the indicator function

$$\mathbf{u}(\mathbf{x}) \equiv \text{swtch}(\mathbf{x} - \mathbf{a}) - \text{swtch}(\mathbf{x} - \mathbf{b}) \quad (\mathbf{a} < \mathbf{b}) \quad (4-11)$$

In the open class interval (a, b) , $u(x) = 1$ and is 0 elsewhere (see also Section 2-8b). We find the desired statistical relative frequency **hh** easily and quickly with **DOT hh = u * 1**.

Statistical relative frequencies can be computed as post-run Monte Carlo statistics. Vectorized Monte Carlo studies can, instead, compute statistical relative frequencies at each sampling point to produce their time histories.

For a given Monte Carlo sample $(x[1], x[2], \dots, x[n])$, the *sample range* **range = xmax – xmin** is the difference between the largest value **xmax** and the smallest value **xmin** in the sample. The DYNAMIC program segment of a *vectorized* Monte Carlo study can compute **xmax**, **xmin**, and **range** at each point of time to produce their time histories. With reference to Section 3-8, we declare an **n**-dimensional vector **xx** and add the DYNAMIC-segment lines

```
Vector xx^ = x      | DOT xmax = xx * 1
Vector xx^ = - x    | DOT mxmin = xx * 1
```

The experiment-protocol script then computes **range = xmax + mxmin**. For repeated-run Monte Carlo simulation, post-run computation of **xmax** and **xmin** requires a search loop in the experiment protocol.

As we already noted, many other statistics (correlation and regression coefficients, and test statistics such as t and χ^2)[5] are functions of sample averages. Post-run estimation of probability densities will be discussed in the next section.

4-9. Post-run Probability-density Estimation [8,9]

(a) A Simple Probability-density Estimate

For continuous random variables **x** the probability density $\phi_x(X)$ for each value **X** of **x** is approximated by

$$\phi_x(X) \approx \text{Prob}\{X - h \leq x < X + h\}/2h = p/2h \quad (4-12)$$

where **2h** is a small class-interval width. For a given Monte Carlo sample $(x[1], x[2], \dots, x[n])$ of **x**-values, we again estimate **p** by the sample average $\langle u(x - X) \rangle$ of an indicator function $u(x - X)$ equal to 1 if $X - h \leq x < X + h$ and 0 otherwise. Specifically, $u(x - X) \equiv \text{rect}((x - X)/h)$, where **rect(x)** is the library function defined in Fig. 2-5c. For small “window widths” **2h** we thus estimate the probability density $\phi_x(X) \approx p/2h$ by

$$f(X) \equiv (1/2h) \langle \text{rect} [(x - X)/h] \rangle \equiv (1/2hn) \sum_{k=1}^n \text{rect} ((x[k] - X)/h) \quad (4-13)$$

For random samples of size n , $2hn$ $f(\mathbf{X})$ has a binomial distribution with success probability p , [4] so that

$$E\{f(\mathbf{X})\} = p/2h, \quad \text{Var}\{f(\mathbf{X})\} = p(1 - p)/4nh^2 \quad (4-14)$$

and for small window widths h

$$E\{f(\mathbf{X})\} \approx \phi_x(\mathbf{X}) \quad \text{Var}\{f(\mathbf{X})\} \approx \phi_x(\mathbf{X})[1 - 2h\phi_x(\mathbf{X})]/2nh \approx \phi_x(\mathbf{X})/2nh \quad (4-15)$$

Because good resolution (small h) implies fewer data points in each window and thus larger estimate variances, *probability-density measurement always involves a compromise between resolution and variance*. You may need a large sample size n .

(b) Triangle and Parzen Windows [8,9]

We usually want to estimate $\phi_x(\mathbf{X})$ for multiple \mathbf{X} -values and would like to fit the estimated $\phi_x(\mathbf{X})$ values with a smooth curve. Estimates of $\phi_x(\mathbf{X})$ for different \mathbf{X} -values separated by less than the window width $2h$ effectively use some sample values more than once. Qualitatively speaking, this means that a curve fitted to the estimate points can be smoothed and appears to exhibit less fluctuation than individual measurements would.

Improved probability-density estimates attempt to enhance this effect. We replace the rectangle-window estimate (21) with the sample average

$$f(\mathbf{X}) \equiv \langle k[(\mathbf{x} - \mathbf{X})/h]/h \rangle \quad (4-16)$$

of a new bump-shaped *kernel function* $k[(\mathbf{x} - \mathbf{X})/h]/h$ centered on the argument \mathbf{X} of the desired estimate $f(\mathbf{X})$. The *window width* h of a kernel determines the spread of the bump and thus the resolution of the probability-density estimate. h can be made smaller for larger sample sizes n . Our primitive rectangular window $\text{rect}[(\mathbf{x} - \mathbf{X})/h]/2$ “weights” all \mathbf{x} -values falling into its window equally and suppresses all others, but more general kernel functions $k[(\mathbf{x} - \mathbf{X})/h]$ let \mathbf{x} -values farther away from the argument value \mathbf{X} contribute to the sample average. Since $\phi_x(\mathbf{X})$ is continuous, this provides a sort of interpolation and may reduce the estimate variance for a given resolution.

The probability-density estimate $f(\mathbf{x})$ is correctly normalized if the kernel function $k(\mathbf{X})$ is normalized, so that

$$\int_{-\infty}^{\infty} k(\mathbf{X}) d\mathbf{X} = 1 \quad \text{implies} \quad \int_{-\infty}^{\infty} f(\mathbf{X}) d\mathbf{X} = 1$$

The estimate mean and variance cannot be derived as easily as in Eq. (14). But it can be shown [10] that $\mathbf{f}(\mathbf{x})$ is an asymptotically unbiased and consistent estimate of $\phi_{\mathbf{x}}(\mathbf{X})$, and

$$\text{Var}\{\mathbf{f}(\mathbf{X})\} \rightarrow (1/nh) \phi_{\mathbf{x}}(\mathbf{X}) \text{KK as } n \rightarrow \infty \quad \text{with} \quad \text{KK} = \int_{-\infty}^{\infty} \mathbf{k}^2(\mathbf{q})d\mathbf{q} \quad (4-17)$$

provided that

$$\begin{aligned} \int_{-\infty}^{\infty} |\mathbf{k}(\mathbf{X})| d\mathbf{x} < \infty, & \quad \sup_{(-\infty, \infty)} |\mathbf{k}(\mathbf{X})| < \infty & \quad \lim_{n \rightarrow \infty} [\mathbf{X}\mathbf{k}(\mathbf{X})] = 0, \\ \lim_{n \rightarrow \infty} h(n) = 0 & \quad \lim_{n \rightarrow \infty} [nh(n)] = \infty \end{aligned}$$

The *rectangle-window* estimate (13) is a special case of the estimate (16); here

$$\mathbf{k}(\mathbf{X}) \equiv \text{rect}(\mathbf{X}/h)/2 \quad \text{with} \quad \text{KK} = 1/2$$

The next-simplest example uses the *triangle-window* kernel

$$\mathbf{k}(\mathbf{X}) \equiv \lim(1 - |\mathbf{X}/h|) \quad \text{with} \quad \text{KK} = 2/3$$

In effect, this mixes \mathbf{x} -values from three neighboring class intervals. But we usually prefer the *Parzen-window* kernel

$$\mathbf{k}(\mathbf{X}) \equiv \exp(-\mathbf{X}^2/2)/\text{sqrt}(2\pi) \quad \text{with} \quad \text{KK} = 1/[2\text{sqrt}(\pi)]$$

which gives some weight to all \mathbf{x} -values. Its resolution-determining window width h measures the spread of a Gaussian-shaped kernel function.

(c) Computation and Display of Parzen Window Estimates

Given a post-run Monte Carlo sample (vector) $\mathbf{x} \equiv (\mathbf{x}[1], \mathbf{x}[2], \dots, \mathbf{x}[n])$, the Parzen-window estimate of the probability density $\phi_{\mathbf{x}}(\mathbf{X})$ is the average $\mathbf{F}(\mathbf{X})$ of the n sample values

$$\text{ff}[i] = \exp[-(\mathbf{X} - \mathbf{x}[i])^2/2 h^2] / [h * \text{sqrt}(2\pi)] \quad (i = 1, 2, \dots, n) \quad (4-18)$$

For each given sample size n , our choice of the window width h will be a trial-and-error compromise between the \mathbf{X} resolution and the smoothness of the estimated probability-density curve. To use smaller window widths h , n has to be increased.

For post-run estimation of probability densities, we add the lines

```

ARRAY f[n] | -- declare a vector of sample values f[i] = ff[i]/n
irule 0 | -- this DYNAMIC segment handles only sampled data
t = 0 | TMAX = 1
a = (select the range a = X2 - X1 of the X-sweep)

b = (select the starting value b = X1 of X)
NN = (select the number of estimated values)
h = (select the Parzen-window width)
alpha = 1/(2 * h^2) | beta = 1/(h * n * sqrt(2 * PI))
drun PARZEN

```

at the end of our experiment-protocol script, say of Figure 4-3 or 4-4. These script lines set parameter values and then execute an extra DYNAMIC program segment labeled **PARZEN**, which will produce probability-density estimates **F(X)** for **NN** values of **x** between **x = X1** and **X = X2** as **t** increases from **t = 0** to **t = TMAX = 1**. Note that the values selected for **NN**, **scale**, and **TMAX** are different from the values used for the simulation itself.

The extra DYNAMIC program segment computes and averages the expression (4-18) to produce **NN** values of **F(X)** between **X = t0** and **X = t0 + TMAX**:

```

label PARZEN
--
X = a * t + b | -- this sweeps X from X1 to X2 as t increases
-- compute n samples ff[i] = f[i]/n
Vector f = beta * exp(- alpha * (X - x)^2)
DOT F = f * 1 | -- sum to average (note that 1/n is included in beta)
dispxy X, F

```

Note that **x** is a vector of sample values, and **X** is a scalar. The last line plots **F(X)** versus **X**. Since **F** is always positive, we usually display **FF = c * F - scale**, where **c** is a scale factor. Figure 5-2c shows such a probability-density display. The Parzen-window technique can be extended to multidimensional probability distributions (Fig. 4-5).

4-10. Combining Vectorized and Repeated-run Monte Carlo Simulation

Since model replication effectively multiplies the number of state variables by **n**, a simulation problem with many differential-equation state variables may not fit a single vectorized Monte Carlo run.⁶

⁶ Currently, DESIRE admits up to 40,000 (Linux) or 20,000 (Windows) differential-equation state variables with Euler and fixed or variable-step Runge-Kutta integration (**irule** 2–7), or up to 600 state variables with more advanced variable-step/variable-order integration (**irule** 9–16). But realistic simulations can involve over 100 differential equations, and we may want large sample sizes **n**.

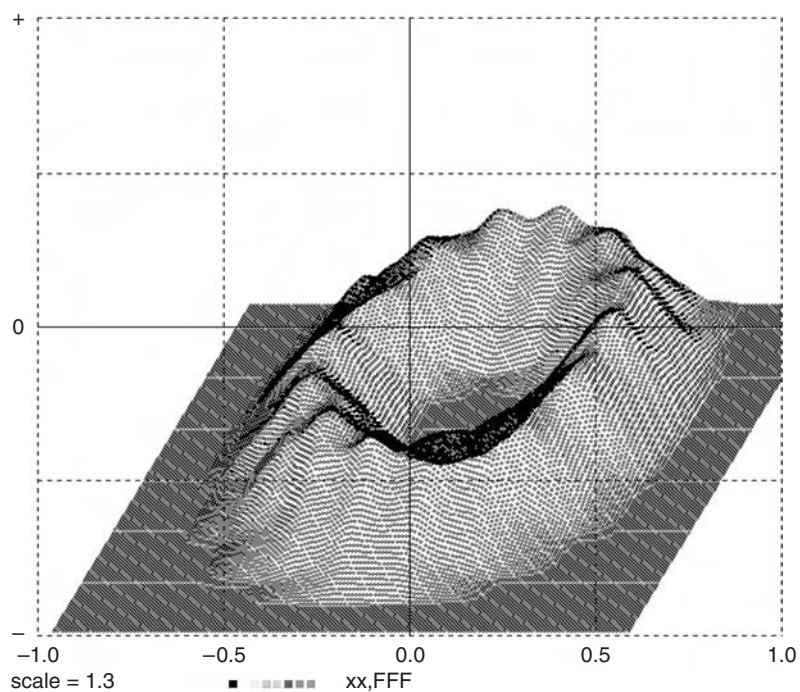
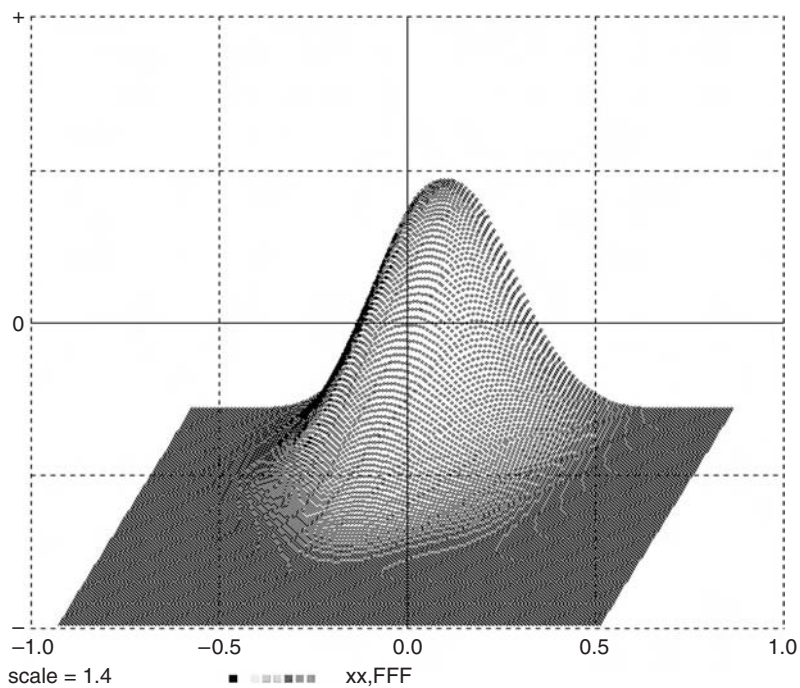


FIGURE 4-5. Two-dimensional Parzen-window probability-density estimates **Fxy** obtained with
Vector fxy = gamma * exp(- alpha * ((y - X)^2 + (y - Y)^2))
DOT Fxy = fxy * 1
 (based on Reference [9]).

This problem is easily solved: we simply repeat vectorized Monte Carlo runs. **nn** repetitions of an **n**-dimensional vectorized simulation result in the overall sample size **M = n * nn**. Figure 4-6 shows the experiment-protocol script for a Monte Carlo study that loops to perform **nn** vectorized runs of our cannonball simulation. Each run again exercises **n** replicated models.

```
--      REPEATED/VECTORIZED MONTE CARLO SIMULATION
-----
DT = 0.008 | TMAX = 50 | N = 5000 | scale=5000
-----
R = 7.5E-05 | g = 32.2
v0 = 900 | -- muzzle velocity
a = 0.03 | -- noise amplitude
--
n = 10 | nn = 2 | M = n * nn | -- sample size
STATE x[n], y[n], xdot[n], ydot[n]
ARRAY theta0[n], v[n], xImpact[n]
--
ARRAY xl[M] | -- combined-sample array
--
for k = 1 to nn | -- nn vectorized simulation runs
  --
  for i = 1 to n | -- noisy elevation angle, radians
    theta0[i] = 70 * PI/180 + a * (ran()+ran()+ran()+ran())
    xdot[i] = v0 * cos(theta0[i]) | ydot[i] = v0 * sin(theta0[i])
    xImpact[i] = 0 | -- reset difference-equation state variable!
  next
  -- | make a vectorized simulation run and read its
  drunr | n sample values into the combined sample
  --
  for i = 1 to n | xl[i + (k - 1) * n] = xImpact[i] | next
next
--
DOT xSum = xl * 1 | xAvg = xSum/M
DOT xxSum = xl * xl | xxAvg = xxSum/M
s=sqrt(abs(xxAvg - xAvg^2))
write "xAvg = ";xAvg," s = ";s | -- the resulting statistics!
```

FIGURE 4-6. Commented experiment-protocol script for **nn** repetitions of the **n**-dimensional vectorized Monte Carlo simulation in Section 4-11. Note that the **n** difference-equation state variables **xImpact[i]** must be reset to 0 for repeated runs.

To compute statistics, we declare an **M**-dimensional combined-sample vector for each random variable of interest, say

```
M = n * nn      |      ARRAY xl[M]
```

for the variable **xlImpact**. After each vectorized run, the resulting **n** sample values **xlImpact[1]**, **xlImpact[2]**, ..., **xlImpact[n]** are fed to the combined-sample vector **xl** with a small one-line loop

```
for i = 1 to n | xl[i + (k - 1) * n] = xlImpact[i] | next
```

The **M**-dimensional combined-sample array **xl** will then produce Monte Carlo statistics such as averages and probability-density estimates exactly as in Sections 4-8 to 4-10.

REFERENCES

1. <http://www.cooper.edu/engineering/chemechem/MMC/tutor.html> presents an excellent review of large-scale Monte Carlo simulation.
2. G. A. Korn, *Interactive Dynamic System Simulation with Microsoft Windows*, Taylor and Francis, London, 1998.
3. M. Galassi et al., *Reference Manual for the GNU Scientific Library (gsl)* (current edition), <ftp://ftp.gnu.org/gnu/gsl/> (printed copies can be purchased from Network Theory Ltd. at <http://www.network-theory.co.uk/gsl/manual/>).
4. J. L. Melsa and A. P. Sage, *An Introduction to Probability and Random Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1971.
5. G. A. Korn and T. M. Korn, *Mathematical Handbook for Scientists and Engineers*, revised edition, Dover, New York, 2000.
6. G. A. Korn and J. V. Wait, *Digital Continuous-System Simulation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
7. G. A. Korn, *Random-Process Simulation and Measurements*, McGraw-Hill, New York, 1966.
8. K. Fukunaga, *Introduction to Statistical Pattern Recognition*, Academic Press, New York, 1972.
9. G. A. Korn, Measurement of Probability Density, Entropy, and Information in Interactive Monte Carlo Simulation, in *Proceedings of the SCS MULTICON*, San Diego, CA, 1999.

10. G. A. Korn, Model replication techniques for parameter-influence studies and Monte Carlo simulation with random parameters, *Math and Computers in Simulation*, **67**: No.6, 2004, pp. 501–513.
11. G. A. Korn, Fast Monte Carlo simulation of noisy dynamic systems on small digital computers, *Simulation News Europe*, December 2002.
12. G. A. Korn, Real statistical experiments can use simulation-package software, *Simulation Practice and Theory*, **13**, 2005, pp. 39–54.

5

Random-process Simulation and Monte Carlo Studies with Noisy Signals

COMPUTER MODELS OF NOISE PROCESSES

5-1. Noise in DYNAMIC Program Segments

Chapter 4 described how experiment-protocol scripts use the library function **ran()** to produce random parameters and random initial conditions. More general random-process models also need random functions of the time in DYNAMIC program segments. Modeling continuous noise on digital computers raises problems because (1) pseudorandom noise is inherently discontinuous, and (2) models of wideband noise need very large numbers of pseudorandom-noise samples. We will first study sampled-data processes and then go on to continuous-noise models.

5-2. Sampled-data Random Processes

(a) A Platform for Sampled-data Experiments

DYNAMIC program segments without differential equations (no **d/dt** or **Vectr d/dt** statements) implement only sampled-data assignments, including difference-equation systems (Section 2-1). Unless the experiment-protocol

script says otherwise,¹ $t = t0$ defaults to $t = 1$, and **TMAX** defaults to $NN - 1$, so that **COMINT** = 1. Now, t simply counts time steps as it takes the successive values $t = 1, 2, \dots, NN$.

DYNAMIC program segments can freely employ **ran()** in sampled-data assignments such as

p = alpha * ran() + q

This calls **ran()** at each sampling time and generates a noisy sequence **p(1)**, **p(2)**, ..., say as an input to a difference-equation system (Section 2-1). **ran()** works equally well in vector or matrix assignments (Chapter 3), say

Vector v = A * cos(w * t) + B * ran()

Vector and matrix assignments effectively call **ran()** repeatedly to generate successive noisy array components.

This is a readymade system for studying discrete-step random processes. It is, in particular, a useful platform for neural-network simulation (Chapter 6). Difference-equation systems with pseudorandom inputs can model a wide variety of discrete-step random processes, including Markov processes. This large subject must be left for another book, but we will exhibit a familiar example.

(b) A Sampled-data Random Process Model: Coin Tossing

The coin-tossing function

$$x = \text{sgn}(\text{ran}() - a) \quad (0 \leq a \leq 1) \quad (5-1)$$

models coin tosses at successive sampling points if $x = 1$ is interpreted as heads and $x = -1$ as tails. The probability of coming up heads is $P = (1 - a)/2$, so that $a = 1 - 2P$.

(c) Recursive Sampled-data Addition and Time Averaging

The DESIRE program

NN = 10 | drun

DYNAMIC

x = f(t) | xsum = xsum + x

type x, xsum

¹ The experiment protocol can set $t = t0 = 0$ if preferred.

computes successive sums $\mathbf{xsum}(t) = \mathbf{x}(1) + \mathbf{x}(2) + \dots + \mathbf{x}(t)$ for $t = 1, 2, \dots$ in the manner of Section 2-2. As noted there, DESIRE automatically recognizes $\mathbf{xsum} = \mathbf{xsum} + \mathbf{x}$ as a difference equation and assigns the default initial value $\mathbf{x}(0) = 0$, so that we correctly implement the recursion

$$\mathbf{xsum}(t) = \mathbf{xsum}(t - 1) + \mathbf{x}(t) \quad (t = 1, 2, \dots)$$

even though t never actually takes the value $t = 0$.

The sampled-data time average

$$\mathbf{xavg} = \mathbf{xsum}(t)/t = [\mathbf{x}(1) + \mathbf{x}(2) + \dots + \mathbf{x}(t)]/t \quad (t = 1, 2, \dots)$$

can also be computed with a recursive sampled-data assignment

$$\mathbf{xavg} = \mathbf{xavg} + (\mathbf{x} - \mathbf{xavg})/t \quad (5-2)$$

DESIRE again recognizes this as a difference equation (Section 2-2) and automatically assigns the default initial value $\mathbf{xavg}(0) = 0$, so that we correctly implement

$$\mathbf{xavg}(t) = \mathbf{xavg}(t - 1) + [\mathbf{x}(t) - \mathbf{xavg}(t - 1)]/t \quad (t = 1, 2, \dots)$$

For $t = 1$, $\mathbf{xavg}(t - 1) = \mathbf{xavg}(0) = 0$, so that $\mathbf{xavg}(1) = \mathbf{x}(1)$, as it ought to be. One can similarly compute sample averages \mathbf{favg} of any function $\mathbf{f}(\mathbf{x})$ of \mathbf{x} by programming

$$\mathbf{favg} = \mathbf{favg} + (\mathbf{f}(\mathbf{x}) - \mathbf{favg})/t \quad (5-3)$$

Note also that DESIRE programs can use recursive averaging to process real data, say, from files, as easily as computer-generated data [1].

5-3. Modeling Continuous Noise

(a) Deriving "Continuous" Noise from Periodic Pseudorandom Samples

Correct modeling of noisy time functions is more difficult than noisy-parameter generation. Normally, the pseudorandom noise function **ran()** cannot be used directly in differential-equation systems, because

- **ran()** necessarily changes in discrete steps and would compromise numerical integration (Section 2-3).
- Noise must be derived from periodic samples to produce predictable noise power spectra.

Therefore, in properly written DYNAMIC program segments that include differential equations, **ran()** normally appears only in sampled-data assignments

following an **OUT** or **SAMPLE m** statement (Sections 1-8 and 2-3). This ensures periodic sampling at the sampling rate

$$SR = (NN - 1)/TMAX \quad \text{or} \quad SR = (NN - 1)/(m * TMAX) \quad (5-4)$$

SR is determined by the values of **NN**, **TMAX**, and **m** set by the experiment protocol (Section 1-6).²

Numerical integration updates differential-equation system variables in small steps to model continuous or “analog” functions of the time **t** (Sections 1-6 and 1-7). Noisy sampled-data variables fed to a differential-equation system are sample/hold state variables (Section 2-3)³ and thus discontinuous step functions of the time **t**. Such variables are read at every derivative call, but they change only at sampling times, so that numerical integration will be correct. As we noted in Section 2-3, sample-hold state variables require initialization at **t = t0**.

To model continuous noise **Noise = Noise(t)**, we feed a noisy sampled-data state variable **y** to a differential equation system representing a low-pass or band-pass filter, as in

d/dt Noise = - w * Noise + y | -- one-stage low-pass noise filter

.....

OUT

y = a * (ran()+ran()+ran()+ran()) + b | -- y is roughly Gaussian

The noise frequency spectrum is determined by the noise-sampling rate (5-4) and the filter transfer function [2].

Many kinds of random processes can be derived from such simulated “analog” noise, for example,

q = A * sin(w * t) + c * p (sinusoid with additive noise)

q = A * p * sin(w * t) (random-amplitude sinusoid)

q = A * sin(w * t + p) (random-phase sinusoid)

Note that multiple independent noise generators require separate calls of **ran()**. To get partially correlated noise samples **y**, **z**, one can use assignments such as

y = ran() z = ran() + b * y with **E{y * z} = b/3**

² This should be noted if **TMAX** is to be changed, for noise spectra will change unless **NN** is changed as well. If noise sampling faster than the input/output sampling rate is required, set **MM > 1** (Section 1-6).

³ Just as in Figure 2-2, one cannot observe the sample-hold action on a display unless the variable is created following a **SAMPLE m** statement with **m > 1**.

(b) ‘Continuous’ Time Averages

To produce the time average

$$\text{xavg} = (1/t) \int_0^t \text{x} dt \quad (5-5)$$

we program the DYNAMIC segment line

$$d/dt \text{xxx} = \text{x} \quad | \quad \text{xavg} = \text{xxx}/t \quad \text{with} \quad \text{xxx}(0) = 0 \quad (\text{default value}) \quad (5-6a)$$

or

$$d/dt \text{xavg} = (\text{x} - \text{xavg})/t \quad \text{with} \quad \text{xavg}(0) = \text{x}(0) \quad (5-6b)$$

Instead of using the default value $t = t_0 = 0$, we set $t = t_0 = 1.0e-275$ to eliminate the singularity (see also Section 5-2c).

5-4. Problems with Simulated Noise

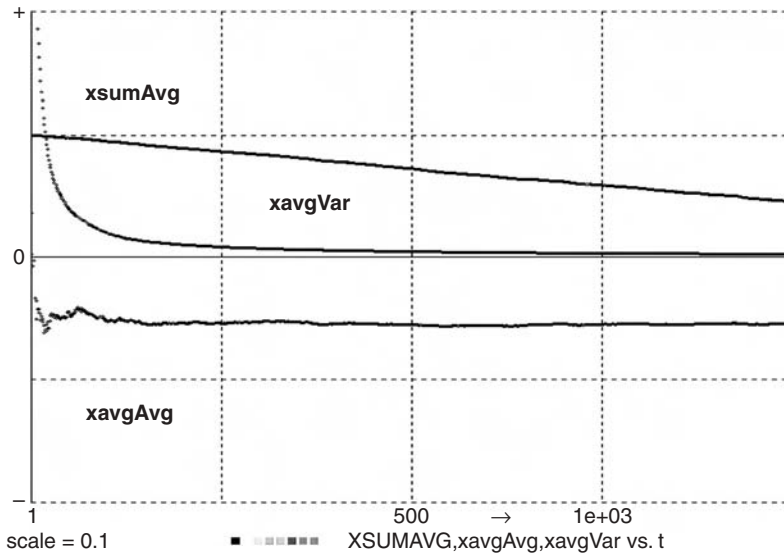
Simulation programs assume that different calls on a pseudorandom-noise generator such as **ran()** produce statistically independent samples of simulated random parameters and noise functions [3–8]. But this is really not true. Pseudorandom-noise samples, although usually guaranteed to be uncorrelated, are generated by a deterministic program. Model outputs can depend on higher-order joint probability distributions of many random-noise samples [6], and it is conceivable that hidden periodicities or correlations might produce strange unforeseen effects. In Chapter 4, our Monte Carlo studies involved only random parameters and/or random initial values; that requires relatively few noise samples and is usually safe. But simulations involving wideband time-variable noise may require enormous numbers of independent noise samples [9]. One thousand simulation runs with, say, 5 noise sources might need 5–500 million independent samples.⁴ References [3] and [5] list a number of tests for the quality of pseudorandom noise, but we usually assume statistical independence and then compare results obtained with different pseudorandom-noise generators. Section 5-10 describes a simple method that completely rescrambles an existing noise sequence for such tests.

MONTE CARLO SIMULATION WITH NOISY SIGNALS

5-5. Gambling Returns

The vectorized Monte Carlo program in Figure 5-1 takes statistics on a sample of n gambling sprees. Each gambling spree consists of $t = N$ successive

⁴ **ran()**, which is based on the GNU library routine **drand48**, repeats after $2^{48} - 1$ samples and normally produces good results. If desired, it would not be difficult to implement **ran()** with a pseudorandom-noise generator having a longer repetition period [3, 4]



```

NN = 1000 | scale = 0.1
display N16 | display C17 | display R
--          P = (1 - a)/2 or a = 1 - 2 * P
P = 0.4857 | --      (as in U.S. roulette)
a = 1-2*P

```

```

-----
n = 1000
ARRAY x[n], xsum[n], xavg[n]
for i = 1 to n | x[i] = sgn(ran() - a) | next | -- initialize
drun

```

```

-----
DYNAMIC

```

```

-----
Vector x = sgn(ran() - a)
Vectr delta xsum = x | Vectr delta xavg = (x - xavg)/t
--

```

```

-----
statistics
DOT temp = xsum * 1 | xsumAvg = temp/n
DOT temp = xsum * xsum | xssAvg = temp/n
xsumVar = xssAvg - xsumAvg^2
DOT temp = xavg * 1 | xavgAvg = temp/n
DOT temp = xavg * xavg | xaaAvg = temp/n
xavgVar = xaaAvg - xavgAvg^2

```

FIGURE 5-1. This vectorized Monte Carlo study computes runtime histories of statistics taken over $n = 1000$ gambling sprees as the number t of bets increases from 1 to $NN = 1000$ bets. Vector difference equations produce the current total-return vector $\mathbf{xsum} = \mathbf{xsum}(t)$ and the current average-return vector $\mathbf{xavg} = \mathbf{xavg}(t)$ by recursive substitutions much as in Section 5-2c.

coin tosses or roulette bets on black/red,

$$\mathbf{x} = \text{sgn}(\text{ran}() - \mathbf{a}) \quad (0 \leq \mathbf{a} \leq 1) \quad (5-7)$$

as in Section 5-2b. We used the theoretical US roulette success probability $\mathbf{P} = 34/(34 + 36) \approx 0.4857$, so that $\mathbf{a} = 1 - 2\mathbf{P} \approx 0.02857$ (\mathbf{P} would be $35/(35 + 36) \approx 0.4923$ in Monte Carlo, Monaco).

For any one gambling spree, the total return $\mathbf{xsum}(\mathbf{t})$ and the average return $\mathbf{xavg}(\mathbf{t})$ after $\mathbf{t} = 1, 2, \dots$ bets are sampled-data variables. Their values would be accumulated by recursive substitutions (difference equations)

$$\mathbf{xsum} = \mathbf{xsum} + \mathbf{x} \quad \mathbf{xavg} = \mathbf{xavg} + (\mathbf{x} - \mathbf{xavg})/\mathbf{t}$$

starting with zero initial values, as in Section 5-2b. For a sample of \mathbf{n} gambling sprees, the experiment protocol in Figure 5-1 declares the total-return vector \mathbf{xsum} and the average-return vector \mathbf{xavg} , whose respective components $\mathbf{xsum}[\mathbf{i}]$ and $\mathbf{xavg}[\mathbf{i}]$ represent the total and average returns of the \mathbf{i} th gambling spree after \mathbf{t} bets. The initial values of $\mathbf{xsum}[\mathbf{i}]$ and $\mathbf{xavg}[\mathbf{i}]$ (for $\mathbf{t} = 0$, as in Section 5-2) all default to 0.

The DYNAMIC program segment in Figure 5-1 accumulates the vector sample functions $\mathbf{xsum} = \mathbf{xsum}(\mathbf{t})$ and $\mathbf{xavg} = \mathbf{xavg}(\mathbf{t})$ after successive bets with the vector difference equations

$$\text{Vectr delta } \mathbf{xsum} = \mathbf{x} \mid \text{Vectr delta } \mathbf{xavg} = (\mathbf{x} - \mathbf{xavg})/\mathbf{t}$$

(see also Section 3-4). Fast compiled DOT operations then generate the sample averages $\mathbf{xsumAvg}(\mathbf{t})$, $\mathbf{xavgAvg}(\mathbf{t})$ and the sample variances $\mathbf{xsumVar}(\mathbf{t})$, $\mathbf{xavgVar}(\mathbf{t})$ for $\mathbf{t} = 1, 2, \dots, \mathbf{NN}$, just as in Section 4-8b. The runtime display in Figure 5-1 shows these statistics as functions of \mathbf{t} . We can also display or print their final values after $\mathbf{t} = \mathbf{NN}$ bets.

Statistics computed from the simulated random process fluctuate less and less as the sample size \mathbf{n} increases (see also the footnote to Section 4-4). For $\mathbf{t} = \mathbf{NN} = 1000$, typical simulation runs with $\mathbf{n} = 1000$ and $\mathbf{n} = 10,000$ produced

$$\begin{array}{ll} \mathbf{xsumAvg} = -29.86 \text{ and } -28.76 & \mathbf{xavgAvg} = -0.02986 \text{ and } -0.0288 \\ \mathbf{xsumVar} = 1073.7 \text{ and } 1008.8 & \mathbf{xavgVar} = 0.0010737 \text{ and } 0.0010088 \end{array}$$

For comparison, probability theory predicts the theoretical values

$$\begin{array}{ll} E\{\mathbf{xsum}\} = -\mathbf{NN} \mathbf{a} = -28.57 & E\{\mathbf{xavg}\} = -\mathbf{a} \approx -0.02857 \\ \text{Var}\{\mathbf{xsum}\} = \mathbf{NN} (1 - \mathbf{a}^2) = 999.184 & \text{Var}\{\mathbf{xavg}\} = (1 - \mathbf{a}^2)/\mathbf{NN} = 0.000999184 \end{array}$$

5-6. A Continuous Random Walk

As a first example of a differential-equation system with dynamic noise input, we generate a continuous random walk in the \mathbf{x} direction by simple integration of a noise input from $\mathbf{t} = \mathbf{t0} = \mathbf{0}$ to $\mathbf{t} = \mathbf{TMAX}$ [10]. A single random walk would be modeled with the simple DYNAMIC program segment

```
DYNAMIC
-----
d/dt x = noise
OUT
noise = a * ran()
```

For $\mathbf{t} = \mathbf{t0} = \mathbf{0}$, all sample values $\mathbf{x}[\mathbf{i}]$ default to 0. Unfiltered, uniformly distributed noise $\mathbf{a} * \mathbf{ran}()$ is obtained from the pseudorandom-noise generator $\mathbf{ran}()$; the positive parameter \mathbf{a} represents the noise amplitude. Since the integrator input **noise** is constant over each sampling interval, it makes sense to select the simple Euler integration (**irule 2**) with $\mathbf{DT} = \mathbf{TMAX}/\mathbf{NN}$ and $\mathbf{t0} = \mathbf{0}$. The “continuous” variable \mathbf{x} actually changes in small steps $\mathbf{a} \mathbf{DT} \mathbf{ran}()$. We set $\mathbf{DT} = \mathbf{TMAX}/\mathbf{NN}$ to make \mathbf{DT} a little smaller than $\mathbf{COMINT} = \mathbf{TMAX}/(\mathbf{NN} - 1)$ (Section 1-8).

Random-walk statistics can be related to probability theory. Each random-walk increment $\mathbf{a} \mathbf{DT} \mathbf{ran}()$ is uniformly distributed between $-\mathbf{a} \mathbf{DT}$ and $\mathbf{a} \mathbf{DT}$, with expected value 0 and variance $(\mathbf{a} \mathbf{DT})^2/3$. Different pseudorandom-noise samples are uncorrelated. By the time \mathbf{t} , Euler integration has added $\mathbf{t}/\mathbf{DT} = \mathbf{t} \mathbf{NN}/\mathbf{TMAX}$ uncorrelated increments whose variances simply add, so that

$$E\{\mathbf{x}(\mathbf{t})\} = 0 \quad \text{Var}\{\mathbf{x}(\mathbf{t})\} = (\mathbf{t}/\mathbf{DT}) (\mathbf{a} \mathbf{DT})^2/3 = \text{Var}(\mathbf{t}) \quad (5-8a)$$

$$E\{\mathbf{x}(\mathbf{TMAX})\} = 0 \quad \text{Var}\{\mathbf{x}(\mathbf{TMAX})\} = \mathbf{NN} (\mathbf{a} \mathbf{DT})^2/3 = \text{VAR0} \quad (5-8b)$$

It is convenient to choose $\mathbf{a} = \mathbf{sqrt}(3 \mathbf{NN})$, so that $\mathbf{scale} = \text{VAR0} = \mathbf{TMAX}^2$. As the number \mathbf{t}/\mathbf{DT} of random steps increases, the theoretical probability density of $\mathbf{x}(\mathbf{TMAX})$ becomes approximately Gaussian with mean and variance (5-8b).

Vectorized Monte Carlo simulation estimates $E\{\mathbf{x}(\mathbf{t})\}$ and $\text{Var}\{\mathbf{x}(\mathbf{t})\}$ by the corresponding sample average $\mathbf{xAvg} = \mathbf{xAvg}(\mathbf{t})$ and sample variance $\mathbf{xVar} = \mathbf{xVar}(\mathbf{t})$ obtained from \mathbf{n} replicated random-walk models

```
DYNAMIC
-----
Vectr d/dt x = noise
OUT
Vector noise = a * ran()
```

A single simulation run produces the n time histories $\mathbf{x}(t)$ and also computes time histories of the statistics $\mathbf{xAvg} = \mathbf{xAvg}(t)$ and $\mathbf{xVar} = \mathbf{xVar}(t)$ with

$$\begin{aligned} \text{DOT } \mathbf{xSum} &= \mathbf{x} * 1 \quad | \quad \text{DOT } \mathbf{xxSum} = \mathbf{x} * \mathbf{x} \\ \mathbf{xAvg} &= \mathbf{xSum}/n \quad | \quad \mathbf{xxAvg} = \mathbf{xxSum}/n \quad | \quad \mathbf{xVar} = \mathbf{xxAvg} - \mathbf{xAvg}^2 \end{aligned} \quad (5-9)$$

Figure 5-2a exhibits several random walks, and Figure 5-2b shows scaled time histories of \mathbf{xAvg} , and $\mathbf{xVar}(t)$. Note how $\mathbf{xVar}(t)$ approximates the theoretical value $\mathbf{Var} = t * DT * (a^2)/3$.

An extra DYNAMIC program segment (Section 4-10c) computes a probability density estimate of $\mathbf{x(TMAX)}$ as in Section 4-10c and compares it to the theoretical Gaussian probability density (Fig. 5-2c). Figure 5-3 shows the complete random-walk program.

For a typical run with $NN = 10,000$ steps and $n = 5000$, we measured $\mathbf{xAvg} = 0.008$ and $\mathbf{xVar} = 1.025$, which approximates the theoretical results (27). Monte Carlo results were unchanged with rescrambled pseudorandom noise obtained by changing the value of n from 5000 to 5002 (Section 5-10). On a 2.4-GHz personal computer running Linux, vectorized simulation of 5000 replicated 10,000-step random walks took 4.6 s, and 7.3 s with runtime computation and display of $\mathbf{xAvg}(t)$ and $\mathbf{xVar}(t)$. Five thousand repeated 10,000-step random walks (without runtime statistics computation) took 8.4 s. The post-run probability density estimation required 2.4 s in either case.

5-7. The 1776 Cannonball with Air Turbulence

Referring to Figure 4-2, we can add a random wind force $\mathbf{W}(t)$ to our cannonball simulation in Section 4-6 by changing the DYNAMIC program segment in Figure 4-3 as follows:

DYNAMIC

```
-----
v = sqrt(xdot^2 + ydot^2)
d/dt W = - r * W + noise | --          a simple low-pass filter
d/dt x = xdot | d/dt y = ydot
d/dt xdot = - R * v * xdot + W | --    W is the horizontal wind force
d/dt ydot = - R * v * ydot - g
--
step
xl = xl + swtch(y) * (x - xl) | --      hold the impact abscissa
OUT
noise = b * (ran()+ran()+ran()+ran()) | --  roughly Gaussian noise
```

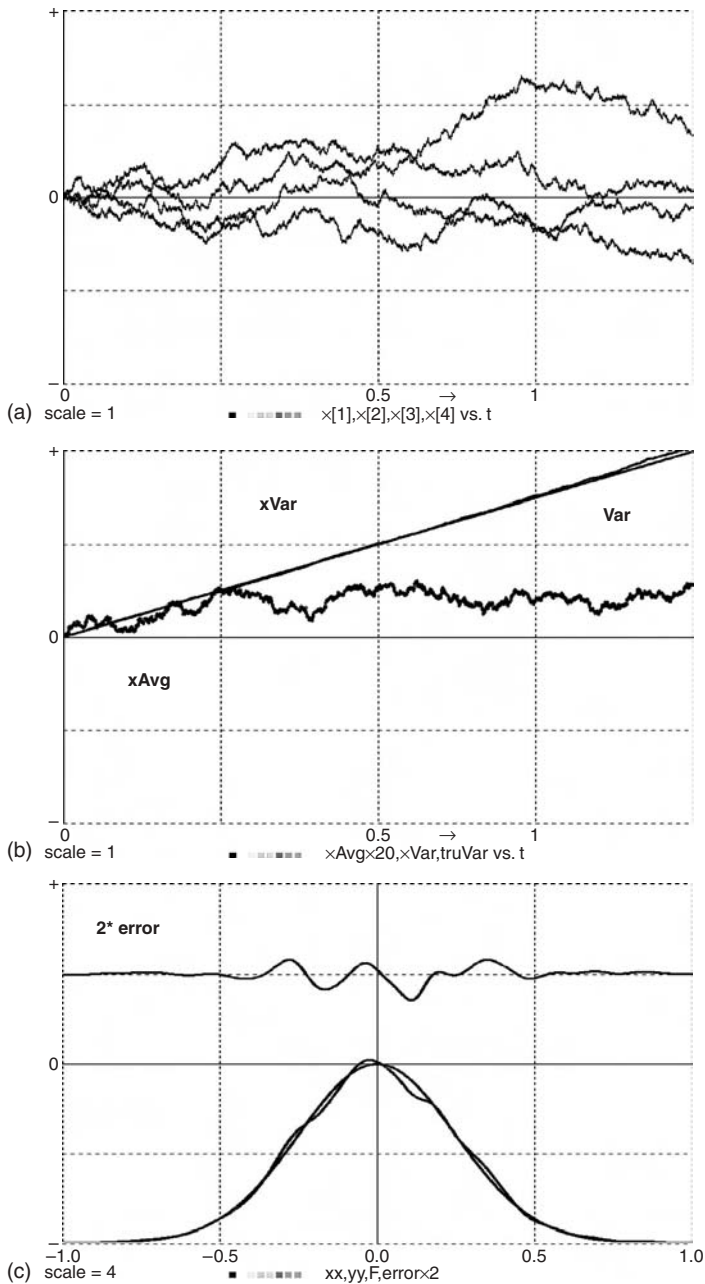


FIGURE 5-2. (a) 4 of $n = 5000$ random walks, (b) time-histories of the statistics $xAvg$ and $xVar$, and (c) post-run probability-density estimate for $x(TMAX)$. Figure 5-2b compares the time history of the sample variance $xVar$ with the theoretical variance $Var = t \Delta t a^2/3$. Figure 5-2c compares the computed probability density estimate with the Gaussian probability density. The original displays were in color.

-- VECTORIZED MONTE CARLO STUDY OF A RANDOM WALK

```

-----
irule 2 | -- Euler integration
NN = 10001 | TMAX = 1
DT = TMAX/NN | -- < COMINT = TMAX/(NN- 1)
a = sqrt(3 * NN) | -- --- scaled noise amplitude
VAR0 = TMAX^2 | scale = VAR0
--
n = 5000 | STATE x[n] | ARRAY noise[n]
for i = 1 to n | noise[i] = a * ran() | next | -- initialize
drun
write "type go to continue" | STOP
-----

-- post-run probability-density estimation
ARRAY f[n]
irule 0 | -- just sampled data
scale = 4 | TMAX = scale | NN = 2500
a = 2 * scale | b = - scale | -- for display sweep
t = 0 | h = 0.15 | -- h is the Parzen-window width
alpha = 1/(2 * h * h) | beta = 1/(h * n * sqrt(2 * PI))
drun PARZEN
-----

DYNAMIC
-----

Vectr d/dt x = noise
OUT
Vector noise = a * ran()
-----
compute statistics
DOT xSum = x * 1 | DOT xxSum = x * x
xAvg = xSum/n | xxAvg = xxSum/n | xVar = xxAvg - xAvg^2
Var = t * DT * (a^2)/3 | -- theoretical variance of x
--
xAvgx20 = 20 * xAvg | -- scaled display
dispt xVar, Var, xAvgx20
-----

label PARZEN
--
xx = a * t + b | -- display sweep for Parzen Window
Vector f = beta * exp(- alpha * (xx - x)^2)
DOT F = f * 1 | F = 10 * F - scale
-- - display Gaussian density for comparison
yy = 10*exp(- (xx^2)/(2 * VAR0))/sqrt(2 * VAR0 * PI) - scale
errorx2 = 2*(F - yy) + 0.5*scale | -- deviation from normal density
dispxy xx, yy, F, errorx2 | -- scaled and offset display

```

FIGURE 5-3. DESIRE program for the vectorized random-walk simulation and runtime statistics computations. An extra DYNAMIC program segment estimates the post-run probability density and compares it with the Gaussian probability density.

The sample-hold state variable **noise** and the extra differential-equation state variable **W** can be safely initialized with zero values. The vectorized model in Figure 4-4 can be similarly amended.

SIMULATION OF NOISY CONTROL SYSTEMS

5-8. Monte Carlo Simulation of a Nonlinear Servomechanism: A Noise-input Test

We use noise-function inputs **unoise(t)** to a simulated control system to study two different problems:

1. How well does the control system follow a deliberately applied random input?
2. How do unwanted noise inputs affect control-system performance?

The following example deals with the first question: we shall employ noise as a test input.

To generate a “continuous” noise test input **unoise(t)** for a simulated control system, we feed roughly Gaussian sample-hold pseudorandom noise **noise = a * (ran()+ran()+ran()+ran())** to a low-pass filter as in Section 5-3a. For a change, let us program a two-section filter:

```
d/dt p = - w * p + noise | --          two-section low-pass filter
d/dt unoise = - w * unoise + p | --    unoise() is the desired test input
.....
OUT | --                               get noise samples at sampling points
noise = a * (ran()+ran()+ran()+ran()) | -- this is roughly Gaussian noise
```

We apply the noise test input **unoise = unoise(t)** to the nonlinear servo model of Section 1-14, that is,

```
e = x - unoise | --          servo error
voltage = - k * e - r * xdot | -- motor voltage
d/dt v = - B * v + voltage | -- motor-field buildup
torque = maxtrq * tanh(g2 * v/maxtrq) | -- saturation-limited motor torque
--
d/dt x = xdot | Vectr d/dt xdot = torque - R * xdot | -- dynamics
```

The Monte Carlo simulation program in Figure 5-4 replicates this model **n** times by declaring state vectors **p**, **unoise**, **v**, **x**, **xdot** and vectors **noise**, **e**,

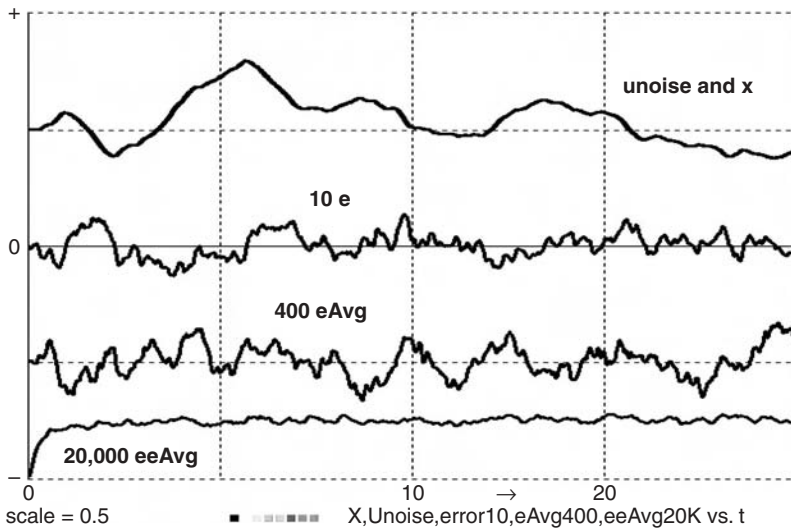


FIGURE 5-4a. This Monte Carlo display shows time histories of 1000 model sample averages **eAvg** and **eeAvg** together with the test-noise input **unnoise[17]** and the corresponding servomechanism output **x[17]** and error **e[17]** for one of the replicated models.

and **torque** with

```
STATE p[n], unnoise[n], v[n], x[n], xdot[n]
ARRAY noise[n], e[n], torque[n]
```

The scalar parameters **w**, **k**, **r**, **B**, **g2**, **maxtorq**, **R**, and **a** are the same for all **n** models. The initial values of **p**, **unnoise**, **v**, **x**, and **xdot** all default to 0. The **n** sample-hold state variables **noise[i]** are initialized with an experiment-control-script loop

```
for i = 1 to n | noise[i] = a * (ran()+ran+ran()+ran()) | next
```

Figure 5-4b displays the resulting time histories of

- The servo input **unnoise[17]** together with the corresponding the servo output **x[17]** and servo error **e[17]** for one of the **n** models.
- The sample average **eAvg = eAvg(t)** of the error.
- The sample average **eeAvg = eeAvg(t)** of the squared error.

After an initial transient, the sample mean square error **eeAvg** exhibits relatively small fluctuations about a fixed expected value. **eeAvg** is a useful statistical control-system performance measure. One can investigate effects of different servo-parameter combinations and also modify the input-noise amplitude and bandwidth by changing **a** and **w**.

-- VECTORIZED MONTE CARLO STUDY OF A NOISE-INPUT TEST**-- note noise sampling and initialization**

```

a = 4 | w = 1
k = 40 | r = 2 | g1 = 10000 | -- controller parameters
B = 300 | maxtrq = 1 | g2 = 2 | R = 0.6 | -- servo parameters

```

```

TMAX = 10 | DT = 0.001 | NN = 5000 | scale = 1
display N1 | display C8 | display R | -- display colors

```

```

-----
n = 1000
STATE p[n], unoise[n], x[n], xdot[n], v[n]
ARRAY noise[n], voltage[n], torque[n], e[n]
-- initialize noise
for i = 1 to n | noise[i] = a * (ran()+ran()+ran()+ran()) | next
--

```

```

drun
write "eAvg = ";eAvg;" eeAvg = ";eeAvg

```

DYNAMIC

```

-----
Vectr d/dt p = - w * p + noise | -- two-section
Vectr d/dt unoise = - w * unoise + p | -- low-pass filter
--

```

```

Vector e = x - unoise | -- servo error
Vector voltage = - k * e - r * xdot | -- motor voltage
Vectr d/dt v = - B * v + g1 * voltage | -- motor-field buildup
--

```

```

Vector torque = maxtrq * tanh(g2 * v/maxtrq) | -- dynamics
Vectr d/dt x = xdot | Vectr d/dt xdot = torque-R * xdot
-----

```

```

-- sample at sampling points

```

OUT

```

Vector noise = a * (ran()+ran()+ran()+ran()) | -- sampled noise
--

```

```

DOT eSum = e * 1 | DOT eeSum = e * e | -- compute averages
eAvg = eSum/n | eeAvg = eeSum/n
-----

```

```

-- offset curves for a rescaled stripchart display
--

```

```

X = x[17] + 0.5 * scale | Unoise = unoise[17] + 0.5 * scale
error5 = 5 * e[17]
eAvg100 = 100 * eAvg - 0.5 * scale
eeAvg500 = 500 * eeAvg - scale
dispt X, Unoise, error5, eAvg100, eeAvg500

```

FIGURE 5-4b. Vectorized Monte Carlo simulation program for the servomechanism noise-input test.

5-9. Monte Carlo Study of Control-system Errors Caused by Noise

In the second type of control system problem, our servomechanism tries to follow a given input $u = u(t)$ such as $u = A * \cos(\omega * t)$ while “continuous” noise $u_{noise}(t)$ is added to the motor voltage $voltage(t)$. We must now follow $u(t)$ as closely as possible and minimize the effect of noise on the control-system output x .

The required simulation program is nearly identical with that in Figure 5-4b. We simply replace the servo input $u_{noise}(t)$ in Figure 5-4b with

$$u = A * \cos(\omega * t)$$

and try to reduce the sample average of the control-system error

$$e = x - u$$

in some sense (Section 5-11). Figure 5-5a lists the program for the vectorized Monte Carlo study. Note that the servo input $u(t)$ and the signal parameters A and ω are common to all n replicated models and are thus represented by scalars. Figure 5-5b shows time histories of $u(t)$, and $u_{noise}[17]$, $x[17]$, and $e[17]$ for one of the models together with the time histories of the sample averages eA_{avg} and eeA_{avg} .

ADDITIONAL TOPICS

5-10. Monte Carlo Optimization

Many Monte Carlo studies are parameter-influence studies (Section 4-3) that attempt to optimize system performance measures defined as sample averages or other statistics. In a control-system study, this could be the sample average of the error at $t = T_{MAX}$, or the sample average of a time integral such as the integral squared error (ISE, Sections 1-14 and 4-3e). Sample averages of time averages computed as in Section 5-2c deserve special mention, because they often have small variances and may thus require smaller Monte Carlo samples.

Vectorization is a convenient and efficient method for computing Monte Carlo sample averages for optimization studies. Unfortunately, though, that is only half the task. Serious parameter optimization typically requires a separate optimization program [15]. Such programs are not trivial and must call the Monte Carlo simulation a number of times—possibly many times. Currently, most such combinations of simulation and optimization are *ad hoc* solutions of special cases.

```

--      VECTORIZED MONTE SIMULATION OF A NOISY SERVO
--      note noise sampling and initialization
-----
A = 0.1 | omega = 1.2 | --                      input-signal parameters
a = 4000 | w = 100 | --                          noise parameters
k = 40 | r = 2 | g1 = 10000 | --                  controller parameters
B = 100 | maxtrq=1 | g2=2 | R=0.6 | -- servo parameters
-----
TMAX = 7.5 | DT = 0.001 | NN = 3750 | scale = 1
display N1 | display C8 | display R | -- display colors
-----
n = 1000
STATE p[n], unoise[n], x[n], xdot[n], v[n]
ARRAY noise[n], voltage[n], torque[n], e[n]
--
-- initialize noise
for i = 1 to n | noise[i] = a * (ran()+ran()+ran()+ran()) | next
--
drun
write "eAvg = ";eAvg;"      eeAvg = ";eeAvg;"
-----
DYNAMIC
-----
Vectr d/dt p = - w * p + noise | --                      two-section
Vectr d/dt unoise = - w * unoise + p | --                low-pass filter
--
u = A * cos(omega * t) | --                      servo input for all n models
Vector e = x - u | --                      servo error
Vector voltage = - k * e - r * xdot + unoise | --        noisy motor voltage
Vectr d/dt v = - B * v + g1 * voltage | --                motor-field buildup
--
Vector torque = maxtrq * tanh(g2 * v/maxtrq) | --        dynamics
Vectr d/dt x = xdot | Vectr d/dt xdot = torque-R * xdot
-----
-- sample at sampling points
OUT
Vector noise = a * (ran()+ran()+ran()+ran()) | --        sampled noise
--
DOT eSum = e * 1 | DOT eeSum = e * e | --        compute averages
eAvg = eSum/n | eeAvg = eeSum/n
-----
-- offset curves for a rescaled stripchart display
--
X = 5 * x[17] + 0.5 * scale
U = 5 * u + 0.5 * scale | Unoise = 0.5 * unoise[17] + 0.5 * scale
error10 = 10 * e[17]
eAvg10 = 10 * eAvg - 0.5 * scale
eeAvg100 = 100 * eeAvg - scale
dispt X, U, Unoise, error10, eAvg1000, eeAvg100

```

FIGURE 5-5a. The vectorized Monte Carlo simulation program for the noise-perturbed servomechanism is similar to Figure 5-4b, but note the different servo input and motor voltage.

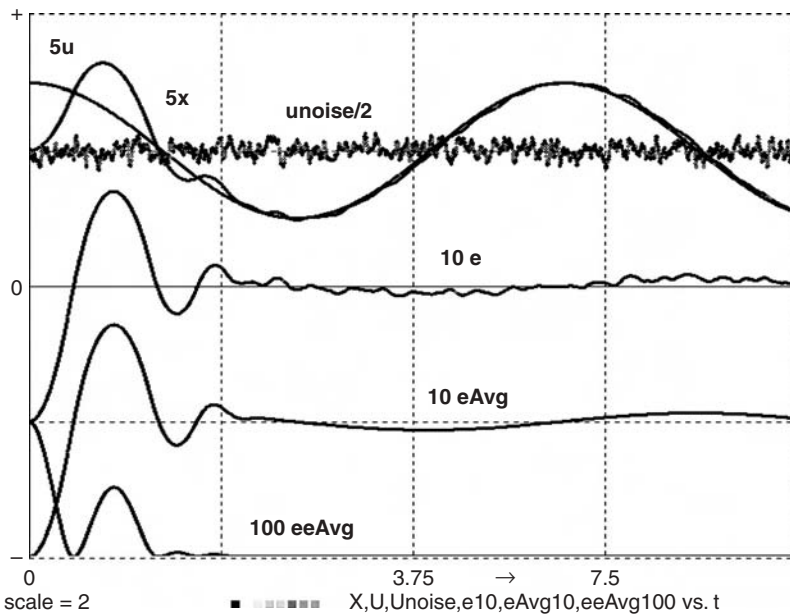


FIGURE 5-5b. Time histories produced by the vectorized Monte Carlo study of a nonlinear servomechanism with a noisy controller. The controller damping coefficient r was deliberately set too low to show the noise effects more clearly. The original display was in color.

5-11. A Convenient Heuristic Method for Testing Pseudorandom Noise

All checks of pseudorandom-noise quality in practical dynamic system simulations are heuristic. But our model-replication technique adds a new simple test to the usual substitution of different noise generators. Since each replicated model is fed its noise in turn, any change in the number n of replicated models completely scrambles the noise sequence fed to each model. Agreement of Monte Carlo results with different values of n , therefore, constitutes a plausible heuristic test of the noise quality.

5-12. An Alternative to Monte Carlo Simulation

(a) Introduction

We showed that respectable Monte Carlo studies of dynamic systems fit on very inexpensive personal computers. But this is a recent development. Monte Carlo simulation of small dynamic systems dates back to the 1940s, but when early guided-missile designers needed to predict mean square errors in noise-perturbed control systems they lacked the computer power needed to simulate

noisy systems repeatedly. They resorted to deriving differential-equation systems whose solution approximated the mean square error directly.

(b) Dynamic Systems with Random Perturbations

We consider differential-equation systems of the form (1-1), say

$$(d/dt)\mathbf{x} = \mathbf{f}[\mathbf{t}; \mathbf{x}, \mathbf{u}] \quad (5-10)$$

where $\mathbf{x} = \mathbf{x}(\mathbf{t}) \equiv (\mathbf{x}_1, \mathbf{x}_2, \dots)$ is a set of state variables, and $\mathbf{u} = \mathbf{u}(\mathbf{t}) \equiv [\mathbf{u}_1(\mathbf{t}), \mathbf{u}_2(\mathbf{t}), \dots]$ represents a set of random input functions and/or system parameters. To simplify the discussion, assume that defined-variable assignments have already been substituted into the state equations (5-10). We again want to study effects of random inputs (noise, wind forces) or parameters (manufacturing tolerances) on the solution time histories $\mathbf{x}(\mathbf{t})$. As before, the initial values $\mathbf{x}(\mathbf{0})$ are simply additional system parameters.

In many applications, each random input \mathbf{u} is the sum $\mathbf{u} = \mathbf{u}_0 + \delta\mathbf{u}$ of a nominal input

$$\mathbf{x}(\mathbf{t}) \equiv \mathbf{x}_0(\mathbf{t}) + \delta\mathbf{x}(\mathbf{t}) \quad (5-11)$$

where $\mathbf{x}_0(\mathbf{t})$ is the nominal solution of the system for $\mathbf{u} = \mathbf{0}$, that is, the solution of

$$(d/dt)\mathbf{x}_0 = \mathbf{f}[\mathbf{t}; \mathbf{x}_0, \mathbf{0}] \quad (5-12)$$

Subtraction of Eq. (5-12) from Eq. (5-10) yields new differential equations for the perturbations $\delta\mathbf{x} = \delta\mathbf{x}(\mathbf{t})$,

$$(d/dt)\delta\mathbf{x} = \mathbf{f}[\mathbf{t}; \mathbf{x}_0(\mathbf{t}) + \delta\mathbf{x}, \mathbf{u}(\mathbf{t}) + \delta\mathbf{u}] - \mathbf{f}[\mathbf{t}; \mathbf{x}_0(\mathbf{t}), \mathbf{0}] \quad (5-13)$$

(c) Mean Square Errors in Linearized Systems

Quite often the perturbations $\delta\mathbf{u}$ and $\delta\mathbf{x}$ are small, so that the perturbation vector $\delta\mathbf{x}(\mathbf{t})$ need not be computed as accurately as $\mathbf{x}(\mathbf{t})$. One can then substitute an accurate solution $\mathbf{x}_0(\mathbf{t})$ of the unperturbed system into Eq.(5-13), which requires less solution accuracy. Specifically, neglecting all but the linear terms in a Taylor-series expansion of Eq. (5-13) produces the linearized perturbation equations

$$d/dt\delta\mathbf{x} = (\partial\mathbf{f}/\partial\mathbf{x})\delta\mathbf{x} + (\partial\mathbf{f}/\partial\mathbf{u})\delta\mathbf{u} \quad (5-14)$$

where the partial derivatives are known functions of the nominal solution $\mathbf{x}_0(\mathbf{t})$ and the time \mathbf{t} but are independent of $\delta\mathbf{x}$ and $\delta\mathbf{u}$.

Control-system designers did not actually need the noisy solutions $\delta\mathbf{x} = \delta\mathbf{x}(\mathbf{t})$ of the perturbation equations (5-14). What they really wanted was a small number of mean-square perturbations

$$\mathbf{XX} \equiv E\{\delta\mathbf{x}^2(\mathbf{t1})\} \quad (5-15)$$

at a specified time $\mathbf{t} = \mathbf{t1}$. Interestingly, it turns out that one can use Eq. (5-14) to derive a new differential-equation system whose solution produces the desired mean squares (5-15) directly; no random-noise input is needed [2,11].

This ingenious approach (originated by Laning and Battin [11]) has been almost forgotten. Straightforward Monte Carlo simulation is no longer expensive, and formulation of the partial derivatives in Eq. (5-14) becomes ugly in flight-simulation problems, where \mathbf{f} involves multi-input tabulated wind-tunnel data.

REFERENCES

1. G. A. Korn, Real statistical experiments can use simulation-package software, *Simulation Practice and Theory*, **13**, 2005, pp. 39–54.
2. G. A. Korn, *Random-process Simulation and Measurements*, McGraw-Hill, New York, 1966.
3. M. Galassi et al., *Reference Manual for the GNU Scientific Library (gsl)* (current edition) (<ftp://ftp.gnu.org/gnu/gsl/>) (printed copies can be purchased from Network Theory Ltd. at <http://www.network-theory.co.uk/gsl/manual/>).
4. P. L'Ecuyer, in *Handbook on Simulation* (Banks et al., eds.), Wiley, New York, 1997, Chapter 4.
5. V. Marsaglia, *Documentation for the DIEHARD Pseudorandom-Noise Test Programs*, <http://stat.fsu.edu/pub/diehard>.
6. P. D. Roberts et al., Statistical Properties of Smoothed Maximal-length Linear Binary Sequences, *Proceedings of IEEE*, January 1966.
7. K. Entacher, On the Cray-system random-number generator, *Simulation*, **72**: No. 3, 1999, pp. 163–169.
8. P. Hellekalek, A note on pseudorandom-number generators, *EUROSIM Simulation News Europe*, July 1997.
9. <http://www.cooper.edu/engineering/chemechem/MMC/tutor.html> presents an excellent review of large-scale Monte Carlo simulation.
10. G. A. Korn, Fast Monte Carlo simulation of noisy dynamic systems on small digital computers, *Simulation News Europe*, December 2002.

11. J. H. Laning and R. H. Battin, *Random Processes in Automatic Control*, McGraw-Hill, New York, 1956.
12. G. S. Fishman, *Monte Carlo Simulation*, Springer, New York, 1995.
13. J. M. Hammersley and D. C. Handscomb, *Advanced Monte Carlo Methods*, Methuen, London, 1964.
14. H. D. Mittelman and P. Spelucci, *Decision Tree for Optimization Software*, <http://plato.asu.edu/guide.html>, 2005.
15. J. J. More and S. J. Wright, *Optimization Software*, SIAM Publications, 1993.

6

Vector Models of Neural Networks

NEURAL-NETWORK SIMULATION

Neural-network simulation began as an attempt to reverse-engineer adaptive biological systems, with neuron pulse rates represented as numerical variables. Connection-parameter adjustments imitate biological learning. Hardware models using hundreds of small processors can implement artificial sense organs, but most artificial neural networks (and all we will discuss here) are abstract *computer models*, in effect large multi-input function generators that compute statistics for regression, pattern recognition, and control systems¹. This chapter is not a treatise on neural-network theory or a roadmap for neural-network development. Our goal here is to demonstrate the use of DESIRE vector operations for programming compact and efficient neural-network models.

6-1. Neural-network Models and Pattern Vectors

Figure 6-1a represents a model neural-network layer with **nx** input activations **x[1], x[2], ..., x[nx]** and **nv** output activations **v[1], v[2], ..., v[nv]**. A simple neuron-layer model implements

$$v[i] = f \left[\sum_{k=1}^{nx} W[i, k]x[k] \right] \quad (i=1, 2, \dots, n) \quad (6-1)$$

¹ Mathematical biologists use more realistic neuron-layer models and now simulate actual pulsing neurons with nonlinear differential equation systems (Sections 6-24 and 6-25).

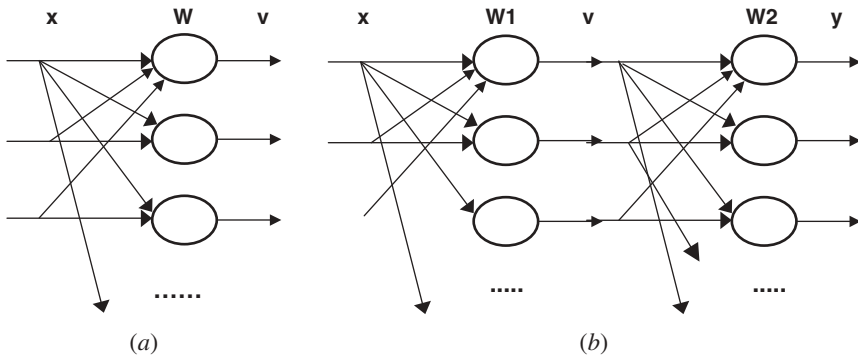


FIGURE 6-1. (a) A neural-network layer, and (b) a two-layer network.

The coefficients $\mathbf{W}[i, k]$ are connection weights and $\mathbf{f}()$ is a neuron activation function. The input and output activations are real-valued features (components) of pattern vectors $\mathbf{x} \equiv (\mathbf{x}[1], \mathbf{x}[2], \dots, \mathbf{x}[\mathbf{nx}])$ and $\mathbf{v} \equiv (\mathbf{v}[1], \mathbf{v}[2], \dots, \mathbf{v}[\mathbf{ny}])$. Typical pattern features are measurement values, image-point intensities, or attributes of an employee, customer, or merchandise item. Pattern vectors admit component-wise addition and multiplication by scalars. Euclidean or taxicab norms $\|\mathbf{x}\|$ of pattern vectors are defined as in Section 3-7b.

Neural-network models combine various types of network layers to produce output patterns as functions $\mathbf{y} = \mathbf{y}(\mathbf{x})$ of an input pattern. We must adjust the connection weights so that the network matches given inputs \mathbf{x} with desired output patterns \mathbf{y} . The neural network is trained for such a task with a training sample of known pairs \mathbf{x}, \mathbf{y} (“supervised learning”). Each training step computes $\mathbf{y}(\mathbf{x})$ and appropriately modifies connection weights as functions of current and past $\mathbf{y}[i]$ values. After the weights are adjusted, we test the neural network with test samples before using it in applications.

DESIRE vector operations produce concise and efficient models for a wide variety of neural networks. The examples in Sections 6-10, 6-12, and 6-22d demonstrate neural-network training. General-purpose simulation-language programs normally train only small neural networks, but our vector models also work with appropriate connection weights precalculated with external optimization programs [3–5].

6-2. Simple Vector Operations Model Neural-network Layers

If the experiment-protocol script declares an \mathbf{nx} -dimensional input-pattern vector \mathbf{x} , an \mathbf{nv} -dimensional output-pattern vector \mathbf{v} , and an $\mathbf{nv} \times \mathbf{nx}$ connection-weight matrix \mathbf{W} with (Section 3-1)

$$\text{ARRAY } x[nx], v[nv], W[nv, nx] \quad (6-2)$$

the basic neural-network layer operation (6-1) is conveniently represented by a DYNAMIC-segment vector assignment (Section 3-3a),

$$\text{Vector } v = f(W * x) \quad (6-3)$$

In particular,

$$\text{Vector } v = W * x \quad (6-4a)$$

represents a linear neural-network layer. The vector assignments

$$\text{Vector } v = \text{SAT}(W * x) \quad \text{Vector } v = \text{sigmoid}(W * x) \quad (6-4b)$$

$$\text{Vector } v = \text{sat}(W * x) \quad \text{Vector } v = \text{tanh}(W * x) \quad (6-4c)$$

model layers of neurons with different output-limiting activation functions (“squashing functions”).² The network layers (6-4b) generate nonnegative outputs $v[i]$ useful for modeling biological pulse rates. Equations (4-4a) and (4-4c) produce more general mathematical models. One can also program scalar assignments to specific individual neuron activations, say $v[13]$.

To add bias inputs $\text{bias}[1]$, $\text{bias}[2]$, ..., $\text{bias}[ny]$ to the neural-network layer (4-3), we declare an nv -dimensional bias vector bias and program³

$$\text{Vector } v = f(W * x + \text{bias}) \quad (6-5)$$

6-3. Normalizing and Contrast-enhancing Neuron Layers

The normalizing and maximum-enhancing layers described in this section are bare-bones abstractions of biologically plausible neuron layers (Section 6-18).

² $\text{sigmoid}(x) \equiv 1/(1 + \exp(-x))$ is a DESIRE library function.

³ It is often convenient to represent bias inputs as connection weights in the $(nx + 1)$ th column of an augmented connection-weight matrix WW . If you declare bias-augmented arrays xx , WW with (Section 3-11)

$$\begin{aligned} \text{ARRAY } x[nx] + x0[1] &= xx, WW[nv, nx + 1] \\ x0[1] &= 1 \end{aligned}$$

one can model the neural-network layer (6-5) with the simpler vector assignment

$$\text{Vector } v = f(WW * xx)$$

The vector expression $WW * xx$ is equivalent to $W * x + \text{bias}$. Note that the true input x is still available to the program, so that a vector expression can be assigned to x . Figures 6-4b and 6-11 show simple applications.

We obtain a *normalized* neuron-layer pattern **v1** with

$$\begin{aligned} \text{Vector } \mathbf{v1} &= \text{abs}(\mathbf{v}) \quad | \quad \text{DOT } \mathbf{vnorm} = \mathbf{v1} * 1 \\ \text{Vector } \mathbf{v1} &= \mathbf{v}/\mathbf{vnorm} \end{aligned} \quad (6-6)$$

(*taxicab normalization*, see also Section 3-7) or

$$\begin{aligned} \text{DOT } \mathbf{vnormsq} &= \mathbf{v} * \mathbf{v} \quad | \quad \mathbf{vnorm} = \text{sqrt}(\mathbf{vnormsq}) \\ \text{Vector } \mathbf{v1} &= \mathbf{v}/\mathbf{vnorm} \end{aligned} \quad (6-7)$$

(*Euclidean normalization*),⁴ so that the activations **v1[i]** or their squares **v1²[i]** add up to 1. Usually the un-normalized vector **v** is not needed, and **v1** in Eq. (6-6) or (6-7) can simply be replaced with **v**. Normalized activations are necessarily bounded.

The output activations **v[i]** of a *softmax* neuron layer defined by

$$\begin{aligned} \text{Vector } \mathbf{v} &= \text{exp}(\mathbf{c} * \mathbf{W} * \mathbf{x}) \quad | \quad \text{-- } (\mathbf{c} > 1) \\ \text{DOT } \mathbf{vsum} &= \mathbf{v} * 1 \quad | \quad \text{Vector } \mathbf{v} = \mathbf{v}/\mathbf{vsum} \end{aligned} \quad (6-8)$$

are normalized and positive. Each **v[i]** is enhanced or reduced depending on how large it is. This contrast enhancement becomes more pronounced as the parameter **c** increases. If no two output activations are equal, the largest **v[i]** approaches 1 as the parameter **c** increases, and all other **v[i]** go to 0. Such a softmax layer is a useful continuous approximation of a normalized maximum-selecting layer defined for the case of all nonnegative **v[i]** by

$$\text{Vector } \mathbf{v}^{\wedge} = \mathbf{W} * \mathbf{x} \quad | \quad \text{Vector } \mathbf{v} = \text{swtch}(\mathbf{v}) \quad (6-9a)$$

(see also Section 3-8b). Another contrast-enhancement technique is *thresholding*, as in

$$\text{Vector } \mathbf{v} = \text{swtch}(\mathbf{c} * \mathbf{W} * \mathbf{x} - \text{thresh}) \quad (\mathbf{c} > 0) \quad (6-9b)$$

where **thresh** is a positive threshold value.

6-4. Multilayer Networks

Assume that the experiment protocol has declared neuron-activation vectors **x**, **v**, **z**, ... and connection-weight matrices **W1**, **W2**, ... with

$$\text{ARRAY } \mathbf{x}[\mathbf{nx}], \mathbf{v}[\mathbf{nv}], \mathbf{z}[\mathbf{nz}], \dots, \mathbf{W1}[\mathbf{nv}, \mathbf{nx}], \mathbf{W2}[\mathbf{nz}, \mathbf{nv}], \dots \quad (6-10)$$

⁴ To save divisions, which are usually slower than multiplications, one can program

$$\text{DOT } \mathbf{vnormsq} = \mathbf{v} * \mathbf{v} \quad | \quad \mathbf{vnormo1} = 1/\text{sqrt}(\mathbf{vnormsq}) \quad | \quad \text{Vector } \mathbf{v1} = \mathbf{v} * \mathbf{vnormo1}$$

Then a DYNAMIC program segment can model a multilayer neural network by simply combining network-layer assignments, as in

$$\begin{aligned} \text{Vector } \mathbf{v} &= \tanh(\mathbf{W1} * \mathbf{x}) \\ \text{Vector } \mathbf{z} &= \mathbf{W2} * \mathbf{v} \\ &\dots\dots\dots \end{aligned} \quad (6-11)$$

The input pattern \mathbf{x} feeds the \mathbf{v} layer, the \mathbf{v} layer feeds the \mathbf{z} layer, and so on (Fig. 6-1b).

6-5. Exercising a Neural-network Model

(a) Computing Successive Neuron Layer Outputs

Neuron-layer definitions such as Eqs. (6-4) to (6-9) are normally sampled-data assignments that execute at the sampling times $\mathbf{t0}$, $\mathbf{t0} + \mathbf{COMINT}$, $\mathbf{t0} + 2 \mathbf{COMINT}$, ..., $\mathbf{t0} + \mathbf{TMAX} = \mathbf{t0} + (\mathbf{NN} - 1)\mathbf{COMINT}$ defined by the experiment protocol (Section 1-6). If an input pattern $\mathbf{x} = \mathbf{x}(\mathbf{t})$ is programmed with a vector assignment such as

$$\text{Vector } \mathbf{x} = \mathbf{A} * \sin(\omega * \mathbf{t}) + \mathbf{a} * \text{ran}() \quad (6-12)$$

then subsequent network-layer assignments such as Eq. (6-11) will generate the neuron-layer outputs $\mathbf{v}(\mathbf{t})$, $\mathbf{z}(\mathbf{t})$, ... for successive sampling times \mathbf{t} . One can now display or list selected neuron activations, say, $\mathbf{v}[\mathbf{19}]$, as functions of the simulation time \mathbf{t} .

$\mathbf{t0}$ and \mathbf{TMAX} default to 1 and $\mathbf{NN} - 1$ if the DYNAMIC program segment does not contain differential equations. If $\mathbf{t0}$ and \mathbf{TMAX} are not specified, then \mathbf{t} simply steps through $\mathbf{t} = 1, 2, \dots, \mathbf{t0} + \mathbf{TMAX} = \mathbf{NN} - 1$.

(b) Using Pattern-row Matrices

Instead of introducing the input pattern as a function of \mathbf{t} as in Eq. (6-12), one can define \mathbf{nx} -dimensional input patterns \mathbf{x} as selected rows of an $\mathbf{N} \times \mathbf{nx}$ pattern-row matrix⁵ \mathbf{P} declared and filled in the experiment-protocol script (Section 6-10a).

After a DYNAMIC program segment specifies the value of the system variable $\mathbf{iRow} > 0$, vector assignments such as

$$\text{Vector } \mathbf{x} = \mathbf{P\#} \quad \text{Vector } \mathbf{x} = (\mathbf{q} - \alpha) * \cos(\mathbf{P\#}) + \mathbf{c} \quad (6-13)$$

⁵ Pattern-row matrices simplify computer programs because almost all computer languages store matrices row-by-row in memory. Pattern vectors, though, are usually represented as column vectors, and most textbooks [15,16] define a pattern matrix as the $\mathbf{nx} \times \mathbf{N}$ matrix \mathbf{X}^T whose columns are our \mathbf{N} \mathbf{nx} -dimensional pattern vectors.

automatically substitute the vector in the (**iRow mod N**)th row of **P** for **P#**. DESIRE returns an error message if **iRow < 1**.

In particular, the DYNAMIC-segment assignment

$$\mathbf{iRow} = t$$

makes **iRow = trunc(t)**, so that the pattern selection cycles through successive rows of **P** for **t = 1, 2, ...**. This allows one to exercise the neural-network model by repeating **N** patterns defined by **P** over and over. Other useful pattern sequences are obtained with

iRow = t/m	(go to next row after m steps)
iRow = k * abs(ran())	(pseudorandom “scrambling” of successive patterns)

DYNAMIC program segments can assign new values to **iRow** as often as needed to produce different pattern sequences. Multiple pattern-row matrices with the same or different dimensions, and with the same or different **iRow** assignments can be used.

P# can be used as a vector in a vector expression as in Eq. (6-11), but must not be index-shifted or used in vector-matrix products.

(c) Pattern Input from Files

For successive presentation of a large number **N** of pattern vectors, one can concatenate the vectors in a file and read successive sets of **N1** vectors into an **N1 × nx** pattern-row matrix **P**. Then, continued simulation runs present **N/N1** successive sets of **N1** patterns to the neural network:

```
connect "datafile" as input #3 | -- open the file
for k = 1 to N/N1
  read #3, P | -- read N1 pattern vectors into the
               pattern-row matrix P
  drun | -- make continued simulation runs with
         the N1 patterns
  next | -- get more patterns
disconnect 3 | -- close the file
```

REGRESSION AND PATTERN CLASSIFICATION

Neural-network models are often applied to two common statistics problems, regression and pattern classification.

6-6. Mean-square Regression

For a sample of corresponding pairs of $\mathbf{n}\mathbf{x}$ -dimensional patterns \mathbf{x} and $\mathbf{n}\mathbf{y}$ -dimensional patterns \mathbf{Y} , mean-square regression of \mathbf{y} on \mathbf{x} produces $\mathbf{n}\mathbf{y}$ -dimensional output patterns $\mathbf{y} = \mathbf{y}(\mathbf{x})$ that minimize the sample average

$$\mathbf{g} = \|\mathbf{y}(\mathbf{x}) - \mathbf{Y}\|^2 \equiv \sum_{k=1}^{\mathbf{n}\mathbf{y}} (\mathbf{y}[\mathbf{k}] - \mathbf{Y}[\mathbf{k}])^2 \quad (6-14)$$

Regression creates a multi-input/multi-output function generator whose output patterns $\mathbf{y}(\mathbf{x})$ match the given samples $\mathbf{Y}(\mathbf{x})$ in the least-squares sense.

6-7. Pattern Classification

Consider a sample of input patterns \mathbf{x} each associated with an prototype pattern taken from a set of $\mathbf{N} \leq \mathbf{n}\mathbf{x}$ known patterns $\mathbf{s} = \mathbf{s}^{(1)}, \mathbf{s}^{(2)}, \dots, \mathbf{s}^{(\mathbf{N})}$. \mathbf{x} is usually simply a noise-perturbed version of \mathbf{s} . A pattern classifier produces output patterns $\mathbf{y} = \mathbf{y}(\mathbf{x})$ that in some sense identify the prototype pattern \mathbf{s} associated with each given input \mathbf{x} . One way to implement this is to do a mean-square regression of \mathbf{s} on \mathbf{x} . But we do not really need to compute an $\mathbf{n}\mathbf{x}$ -dimensional regression function approximating the “best-fitting” prototype pattern $\mathbf{s}^{(i)}$ —all we really want is its index i .

An especially useful computer representation or code for the \mathbf{N} index values $i = 1, 2, \dots, \mathbf{N}$ is the corresponding set of \mathbf{N} \mathbf{N} -dimensional *binary selector patterns*

$$\mathbf{S}^{(1)} \equiv (1, 0, 0, \dots), \mathbf{S}^{(2)} \equiv (0, 1, 0, \dots), \dots, \mathbf{S}^{(\mathbf{N})} \equiv (0, 0, \dots, 1) \quad (6-15)$$

For our sample of paired input patterns \mathbf{x} and corresponding prototypes \mathbf{s} , we will use a classifier network with \mathbf{N} -dimensional output patterns $\mathbf{y}(\mathbf{x})$ that match not \mathbf{s} , but \mathbf{S} , in the least-squares sense. In other words, the $\mathbf{y}(\mathbf{x})$ are to minimize the sample average of

$$\mathbf{g} = \|\mathbf{y}(\mathbf{x}) - \mathbf{S}\|^2 \equiv \sum_{k=1}^{\mathbf{n}\mathbf{y}} (\mathbf{y}[\mathbf{k}] - \mathbf{S}[\mathbf{k}])^2 \quad \text{with} \quad \mathbf{n}\mathbf{y} = \mathbf{N} \quad (6-16)$$

One can show that network output activations $\mathbf{y}[i]$ that minimize the expected value of \mathbf{g} (the “theoretical risk”) equal the *a posteriori* probabilities **Prob** ($\mathbf{s} = \mathbf{s}^{(i)} \mid \mathbf{x}$) [6,7]. The actual network outputs $\mathbf{y}[i]$ are statistical estimates of these *a posteriori* probabilities.

NEURAL-NETWORK TRAINING: PATTERN CLASSIFICATION AND ASSOCIATIVE MEMORY

6-8. Linear Pattern Classifiers

An \mathbf{nx} -dimensional linear neural-network layer

$$\text{Vector } \mathbf{y} = \mathbf{W} * \mathbf{x} \quad (6-17)$$

can classify $\mathbf{N} \leq \mathbf{nx}$ prototype vectors $\mathbf{s}^{(i)}$ if they are linearly independent, that is, if $\mathbf{a}_1 \mathbf{s}^{(1)} + \mathbf{a}_2 \mathbf{s}^{(2)} + \dots + \mathbf{a}_N \mathbf{s}^{(N)} = \mathbf{0}$ implies $\mathbf{a}_1 = \mathbf{a}_2 = \dots = \mathbf{a}_N = \mathbf{0}$. This is true in many applications, for example, for almost all image-pattern prototypes.⁶

The desired optimal connection-weight matrix \mathbf{W} can be computed explicitly if each classifier input \mathbf{x} is simply one of \mathbf{N} linearly independent prototype patterns $\mathbf{s}^{(i)}$ with additive zero-mean noise. \mathbf{W} is then the *Penrose pseudoinverse* of the $\mathbf{nx} \times \mathbf{N}$ pattern matrix \mathbf{X}^T whose \mathbf{N} columns are the given \mathbf{nx} -dimensional prototype vectors.⁷ But the successive-approximation technique described below is simpler, and it is not restricted to additive noise.

6-9. The LMS Algorithm

We start with random connection weights $\mathbf{W}[i, k]$ and feed our one-layer network (6-17) successive noise-perturbed prototype patterns, say $\mathbf{x} = \mathbf{s} + \mathbf{a} * \mathbf{ran}()$. To reduce the error measure (6-16) at each step, *Widrow's LMS algorithm* (least-mean-squares algorithm) or *delta rule* [8,9] repeatedly moves each connection weight $\mathbf{W}[i, k]$ in the negative-derivative direction by assigning

$$\mathbf{W}[i, k] = \mathbf{W}[i, k] - \frac{1}{2} \text{lrate } \partial g / \partial \mathbf{W}[i, k] \quad (i = 1, 2, \dots, \mathbf{N}; k = 1, 2, \dots, \mathbf{nx}) \quad (6-18)$$

where

$$\mathbf{g} = \sum_{i=1}^{\mathbf{N}} \left(\sum_{j=1}^{\mathbf{nx}} \mathbf{W}[i, j] \mathbf{x}(r)[j] - \mathbf{S}[i] \right) \quad (6-19)$$

$$\frac{1}{2} \frac{\partial \mathbf{g}}{\partial \mathbf{W}[i, k]} = \sum_{j=1}^{\mathbf{nx}} (\mathbf{W}[i, j] \mathbf{x}(r)[j] - \mathbf{S}[i] \mathbf{x}(k)) \quad (i = 1, 2, \dots, \mathbf{N}; k = 1, 2, \dots, \mathbf{nx}) \quad (6-20)$$

⁶ Often, prototype vectors can be made linearly independent by adding the same constant vector to every prototype, or by using more pattern components (e.g., by adjoining an extra constant component to every pattern \mathbf{x}).

⁷ Pseudoinverses, and the Greville and Gram-Schmidt algorithms used to compute them, are treated in References [33, 34].

The LMS algorithm (6-18) simplifies computations by using derivatives of \mathbf{g} itself instead of derivatives of its sample average. In effect, the LMS algorithm approximates each derivative of the sample average by accumulating many small steps.

The choice of the optimization gain **lr**ate is a trial-and-error compromise between computing speed and stable convergence. Successive values of **lr**ate must decrease to avoid overshooting the optimal connection weights. More specifically, if the sum of all successive squares **lr**ate² has a finite limit, then the LMS algorithm converges with probability 1 to at least a local minimum of the expected value $E\{\mathbf{g}\}$ (the theoretical risk, as in Section 6-7), assuming that such a minimum exists [8,9]. The algorithm should then approximately minimize measured sample averages of \mathbf{g} (the empirical risk). In any case, results must be checked with multiple samples.

DESIRE's computer-readable vector/matrix language represents the $\mathbf{N} \times \mathbf{nx}$ matrix (6-20) neatly as the outer product $(\mathbf{W} * \mathbf{x} - \mathbf{S}) * \mathbf{x}$ of the \mathbf{N} -dimensional vector $\mathbf{W} * \mathbf{x} - \mathbf{S}$ and the \mathbf{nx} -dimensional vector \mathbf{x} (Section 3-10). We start the connection weights $\mathbf{W}[i, k]$ with random values and implement the LMS algorithm (6-18) with the matrix difference equation

$$\text{MATRIX } \mathbf{W} = \mathbf{W} - \text{lr}ate * (\mathbf{W} * \mathbf{x} - \mathbf{S}) * \mathbf{x}$$

or, more simply,

$$\text{DELTA } \mathbf{W} = \text{lr}ate * (\mathbf{S} - \mathbf{W} * \mathbf{x}) \quad (6-21)$$

6-10. A Softmax Image Classifier

(a) Problem Statement and Experiment-protocol Script

The program in Figure 6-2 models an effective classifier network for 5×5 -pixel image patterns representing the $\mathbf{N} = 26$ letters of the alphabet. Each letter image is an instance of the vector **input**, whose $\mathbf{nx} = 5 \times 5 = 25$ components are pixel-intensity values. We simply used the values -1 for blank pixels and $+1$ for black or colored pixels. Each actual network input \mathbf{x} is such a letter pattern perturbed by additive noise, that is,

$$\mathbf{x}[i] = \text{input}[i] + \text{Tnoise} * \text{ran}() \quad (i = 1, 2, \dots, \mathbf{nx})$$

The experiment protocol declares a pattern-row matrix **INPUT** (Section 6-5b) with $\mathbf{N} = 26$ rows of $\mathbf{nx} = 25$ pixel values, one row for each letter of the

alphabet. A single **data/read** assignment fills this matrix with successive pixel values arranged in 26 groups of 5 5-pixel lines:

```
--                                A
data 1,1,1,1,1
data 1,-1,-1,-1,1
data 1,1,1,1,1
data 1,-1,-1,-1,1
data 1,-1,-1,-1,1
--                                B
data 1,1,1,1,1
data 1,-1,-1,-1,1
data 1,1,1,1,-1
data 1,-1,-1,-1,1
data 1,1,1,1,1
--
..... etc. for C, D, ...
read INPUT
```

The corresponding 26 binary selector patterns **S** = (1, 0, 0, ...), (0, 1, 0, ...), ... similarly form the rows of an **N × N** row-pattern matrix **TARGET**. This is simply the **N × N** unit matrix defined by the DESIRE script line **TARGET = 1**. A double experiment-protocol script loop

```
for i = 1 to N | for k = 1 to nx+1 | W[i, k] = ran() | next | next
```

starts the unknown connection weights with random values.

(b) Network Model and Training

Instead of a simple linear network layer, we use a softmax layer (Section 6-3), whose contrast-enhanced output activations are especially suitable for matching binary-selector patterns. Referring to Sections 6-3 and 6-5b, the DYNAMIC program segment in Figure 6-2 represents the complete neural network with

DYNAMIC

```
-----
iRow = t | Vector x = INPUT# + Tnoise * ran() | -- read one row
Vector v = exp(c * W * x) | DOT vsum = v * 1 | -- softmax
Vector v = v/vsum
```

When the network is trained, the normalized softmax output activations **v[i]** estimate the *a posteriori* probabilities of the 26 alphabet-letter patterns (Section 6-10c).

```

--                SOFTMAX PATTERN CLASSIFIER
--                estimates a posteriori probabilities ...
--                ... and implements associative memory
-----
nx = 25 | N = 26
ARRAY input[nx], x[nx]
ARRAY INPUT[N, nx], TARGET[N, N], W[N, nx]
ARRAY v[N], q[N], y[nx], error[N], Yerror[nx]
--
for i = 1 to N | for k = 1 to nx | -- initialize W
W[i,k] = ran() | next | next
--
-----                input-pattern rows
-- A
data 1,1,1,1,1
data 1,-1,-1,-1,1
data 1,1,1,1,1
data 1,-1,-1,-1,1
data 1,-1,-1,-1,1
-- B
data 1,1,1,1,1
data 1,-1,-1,-1,1
data 1,1,1,1,1
data 1,-1,-1,-1,1
data 1,1,1,1,1
..... etc. for C, D, ...
read INPUT
MATRIX TARGET = 1 | -- binary-classifier rows
lrate = 0.05 | Tnoise = 0.5 | Rnoise = 0.9 | c = 0.1
NN=20000
-----
drun
write 'type go for successive recall runs' | STOP
-----
display F | -- clear the display
t = 1 | NN = 2 | restore | -- reset the read pointer
label recall
for l = 1 to N
read input | drun RECALL
write v | -- show all 26 probabilities
write 'type go for successive recall runs' | STOP
next
restore | go to recall

```

FIGURE 6-2a. This experiment-protocol script for the pattern classifier trains the network with **NN = 20,000** noise-perturbed patterns and then calls a second DYNAMIC program segment for successive tests with each pattern. Each test estimates the *a posteriori* probabilities and displays the actual patterns **input**, **x**, **yy**, and **y** (Fig. 6-2b). Try this program with different noise levels **Tnoise**, **Rnoise**.

DYNAMIC

```

iRow = t | Vector x = INPUT# + Tnoise * ran()
Vector v = exp(c * W * x) | DOT vsum = v * 1
Vector v = v/vsum | -- probability estimate

```

```

Vector error = TARGET# - v
DELTA W = lrate * error * x | -- LMS algorithm
DOT enormsq = error * error

```

```
--
dispt enormsq

```

label RECALL

```

--
Vector x = input + Rnoise * ran()
Vector v = exp(c * W * x) | DOT vsum = v * 1
Vector v = v/vsum | -- probability estimate
Vector q^ = v | Vector q = swtch(q) | -- binary selector
Vector yy = INPUT% * v
Vector y = INPUT% * q | -- associative memory

```

```

-- pattern outputs
Vector input = cc * input | Vector x = cc * x
Vector yy = cc * yy | Vector y = cc * y
SHOW | SHOW input, 5 | SHOW x, 5 | SHOW yy, 5 |
SHOW y, 5

```

FIGURE 6-2b. DYNAMIC program segments for training and testing the pattern classifier. The test segment computes the *a posteriori* probabilities **p** for the current pattern input and also produces associative-memory outputs **yy** and **y**. Statements such as **SHOW x, 5** display a pattern as 5 rows of 5 pixels. **cc** is a color value.

With **iRow = t = 1, 2, ...**, our neural network is repeatedly fed all **N** successive pattern rows, for a total of **NN** training steps. We define **error = S - W * x** rather than **W * x - S** as the pattern-matching-error vector, because this saves programming a possibly large number of leading minus signs. We then implement the LMS algorithm with

```

Vector error = TARGET# - v
DELTA W = lrate * error * x

```

As training proceeds, the program also computes and displays the squared pattern-matching error $\mathbf{g} = \mathbf{enormsq}$ as a function of the trial number \mathbf{t} (Fig. 6-3), with

$$\text{DOT enormsq} = \text{error} * \text{error} \quad | \quad \text{dispt enormsq}$$

(c) Test Runs and A Posteriori Probabilities

To test the classifier, our experiment protocol restores the data/read pointer and uses **data/read** assignments to feed one pattern at a time directly to the neural-network input vector **input**. No pattern matrix is needed. A second DYNAMIC program segment labeled **RECALL** then runs the neural network once ($\mathbf{NN} = 2$) to compute and list the 26 *a posteriori* probability estimates $\mathbf{p[i]}$ for the first alphabet-letter pattern (Fig. 6-3). Test runs are repeated for the other letter patterns.

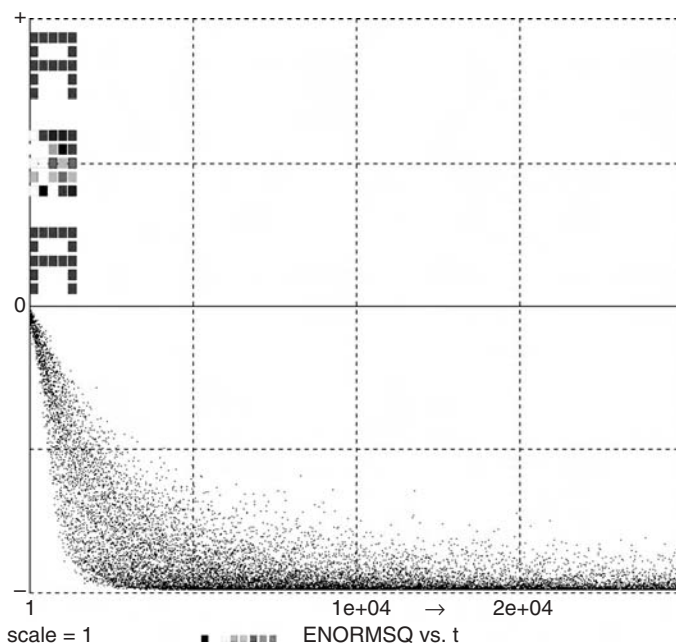


FIGURE 6-3. Referring to the softmax pattern-classifier program in Figure 6-2, the bottom of the display shows superimposed time histories of the squared binary-selector-matching error **enormsq** for all 26 noisy alphabet-letter patterns. The top of the display shows the actual prototype pattern **input**, the noise-corrupted pattern **x**, and the clean associative-memory output **y** for the letter “A”.

6-11. Associative Memory

The **RECALL** program segment in Figure 6-2*b* also computes and displays the **nx**-dimensional pattern **yy** defined by

$$\text{Vector } \mathbf{yy} = \text{INPUT\%} * \mathbf{p} \quad (6-22)$$

Since **p** approximates a binary-selector pattern, **yy** approximates the prototype pattern associated with the noise-perturbed input **x** (associative memory). Our **RECALL** segment also implements a normalized maximum-selecting layer (Section 6-3)

$$\text{Vector } \mathbf{q}^{\wedge} = \mathbf{v} \quad | \quad \text{Vector } \mathbf{q} = \text{swtch}(\mathbf{q}) \quad (6-23)$$

q is necessarily an exact binary-selector pattern, so that the associative-memory output

$$\text{Vector } \mathbf{y} = \text{INPUT\%} * \mathbf{q} \quad (6-24)$$

reproduces a prototype pattern exactly. Strong noise would cause the selection of a wrong prototype, but the system is an effective nonlinear noise filter. Figure 6-3 shows actual patterns **input**, **x**, **yy**, and **y** obtained with moderate noise in the training and recall runs.

NONLINEAR MULTILAYER NETWORKS

6-12. Backpropagation Networks

(a) The Backpropagation Algorithm

We now turn to nonlinear multilayer networks, say a two-layer network⁸ defined by

$$\begin{aligned} \text{Vector } \mathbf{v1} &= \tanh(\mathbf{W1} * \mathbf{x}) \\ \text{Vector } \mathbf{y} &= \tanh(\mathbf{W2} * \mathbf{v1}) \end{aligned} \quad (6-25)$$

The **v1** layer is a hidden layer. For mean-square regression (Section 6-6),⁹ we are given a training sample of paired of **nx**-dimensional patterns **x** and

⁸ Three layers if the input buffer is counted as an extra layer, as many textbooks do.

⁹ Backpropagation networks can also serve as pattern classifiers with a softmax output layer in the manner of Section 6-10 (Example **ex6-1.lst** in the book CD).

ny-dimensional “target” patterns **Y**. We want to generate **ny**-dimensional output patterns **y = y(x)** that minimize the sample average of

$$g = ||y(x) - Y||^2 \equiv \sum_{k=1}^{ny} (y[k] - Y[k])^2 \quad (6-26)$$

Pattern classification can be programmed as mean-square regression on binary selector patterns, with a softmax output layer as in Section 6-10 (example **bpsoft7.lst** in the book CD).

To minimize the mean-square regression error, we update the connection weights **W1[i, k]** and **W2[i, k]** with a generalized version of the LMS algorithm (6-18), the *generalized delta rule*

$$\begin{aligned} W1[i, k] &= W1[i, k] - \frac{1}{2} \text{lr}ate1 \partial g / \partial W1[i, k] \\ &\quad (i = 1, 2, \dots, nv1; k = 1, 2, \dots, nx) \\ W2[i, k] &= W2[i, k] - \frac{1}{2} \text{lr}ate2 \partial g / \partial W2[i, k] \\ &\quad (i = 1, 2, \dots, ny; k = 1, 2, \dots, nv1) \end{aligned} \quad (6-27)$$

This is a system of *difference equations* similar to those in Section 3-1; the connection weights are difference-equation state variables. The right-hand side of each difference equation (6-27) is a function of the *current values* of the connection weights and defined variables (6-25).

We first compute these defined variables. The derivatives in Eq. (6-27) are then evaluated by the chain rule. This rather lengthy derivation [7,8] is simplified if we specify intermediate results in terms of three new defined-variable vectors **error**, **delta1**, and **delta2** declared with

ARRAY error[ny], delta1[nv1], delta2[ny]

In the DYNAMIC program segment, we program

$$\begin{aligned} \text{Vector error} &= Y - y \\ \text{Vector delta2} &= \text{error} * (1 - y^2) \\ \text{Vector delta1} &= W2\% * \text{delta2} * (1 - v1^2) \end{aligned} \quad (6-28)$$

in that order (note that **error** is defined as in Section 6-10), and then update the connection-weight matrices with

$$\begin{aligned} \text{DELTA } W1 &= \text{lr}ate1 * \text{delta1} * x \\ \text{DELTA } W2 &= \text{lr}ate2 * \text{delta2} * v1 \end{aligned} \quad (6-29)$$

lrate1 and **lrate2** are positive, suitably decreasing learning rates similar to **lrate** in Section 6-9. Reference [7] suggests **lrate1** = **r** * **lrate2** with **r** between 2 and 5. Generalization to three or more layers is not difficult.¹⁰ Most backpropagation networks have only two layers, and the output layer is often simply a linear layer.¹¹ Neuron layers can be given bias inputs in the manner of Section 6-2.

Note that **A%** denotes the transpose of a matrix **A** (Section 3-3), products of two vectors represent matrices (Section 3-10), and the function $\mathbf{1} - \mathbf{Q}^2$ is the derivative of the neuron activation function $\mathbf{Q} = \tanh(\mathbf{q})$.

Interestingly, one can consider the defined-variable assignments (6-28) as the definition of a neural network that “backpropagates” output-error effects to the connection weights in each layer of the original neural network.

(b) Discussion

With enough neurons in the first layer, even a two-layer network is theoretically a “universal approximator,” that is, an output activation **y[i]** can approximate any desired continuous function of the inputs **x[k]** [7,8]. Typically, there is more than one optimal set of connection weights. If too many hidden-layer neurons are used, one might match the training-sample pairs accurately but make matches for new test samples worse (“overtraining” hurts “generalization”). This problem is treated in substantial detail in References [11] and [32].

Unfortunately, backpropagation training is often slowed by flat spots and/or stopped by local minima in the mean-square error function. A frequently useful partial fix (momentum learning) declares two extra state-variable matrices

¹⁰ The way to add more layers can be seen from the program for a three-layer network with two hidden layers:

```

Vector v1 = tanh(W1 * x)
Vector v2 = tanh(W2 * v1)
Vector y = tanh(W3 * v2)
Vector error = Y - y
Vector delta3 = error * (1 - y^2)
Vector delta2 = W3% * delta3 * (1 - v2^2)
Vector delta1 = W2% * delta2 * (1 - v1^2)

DELTA W1 = lrate1 * delta1 * x
DELTA W2 = lrate2 * delta2 * v1
DELTA W3 = lrate3 * delta3 * v2

```

Example **bpctest2.lst** in the book CD shows the time histories of some hidden neurons in a three-layer backpropagation network.

¹¹ In this case, the activation-function derivative factor $(\mathbf{1} - \mathbf{y}^2)$ in Eq. (6-28) is omitted, since all its vector components equal 1.

Dw1, Dw2 and replaces the two matrix difference equations (6-29) with four difference equations

$$\begin{aligned} \text{MATRIX Dw1} &= \text{lrate1} * \text{delta1} * \mathbf{x} + \text{mom1} * \text{Dw1} \\ \text{MATRIX Dw2} &= \text{lrate2} * \text{delta2} * \mathbf{v1} + \text{mom2} * \text{Dw2} \\ \text{DELTA W1} &= \text{Dw1} \quad | \quad \text{W2} = \text{Dw2} \end{aligned} \quad (6-30)$$

which make the connection-weight adjustments favor the directions of past successes. The optimization parameters **mom1, mom2** must be found by trial and error and are typically between 0.1 and 0.9. There are literally hundreds of papers and several books [6–9,12–18] describing other improved back-propagation algorithms, but none work every time. References [2–5] describe more advanced numerical function-optimization schemes applicable to multilayer networks. In practice, the best algorithm for a specific application must be selected (and possibly redesigned) by trial and error. The Levenberg–Marquart algorithm [5,7] is often a good compromise.

(c) Examples and Neural-network Submodels

Backpropagation regression networks with a few inputs and one or more outputs are used to model empirical relations. As a simple example, the program in Figure 6-4a trains a two-layer regression network to produce a very accurate sine function; Figure 6-4b shows results. We have defined the two-layer neural network as a reusable submodel (Section 3-17) in the experiment-protocol script. The same submodel is then invoked in two separate DYNAMIC program segments, one for training and one for recall tests. Our submodel could also be stored and used in another program, say, in a control-system simulation.

Figure 6-4c shows the squared-error time histories for 32 output activations of a two-layer, 32-input backpropagation network with **nv** = 9 hidden-layer neurons during a successful training run of **NN** = 200,000 steps. A 2.4-GHz personal computer trained 585 connection and bias weights to produce this display in 7.3 s. The same run took 5.5 s with the display turned off.

6-13. Radial-basis-function Networks

(a) Basis-function Expansion and Linear Optimization

Given a sample of corresponding measurements **x, Y**, traditional statistical regression methods have long approximated mean-square regression functions **y(x)** (Section 6-6) with weighted sums **y(x) = W1 f1(x) + W2 f2(x) + ... + Wn fn(x)** of conveniently chosen basis functions **f1(x), f2(x), ..., fn(x)**. One


```

--                                     A FUNCTION- LEARNING BACKPROPAGATION NETWORK
-----
--                                     note that the submodel definition does not depend on nx, ny, nv
--
ARRAY x$[1], y$[1], v$[1], W1$[1, 1], W2$[1, 1]
SUBMODEL NET2(x$, y$, v$, W1$, W2$)
  Vector v$ = tanh(W1$ * x$)
  Vector y$ = W2$ * v$
end
-----
nx = 1 | ny = 1 | nv = 5 | --                                     nv is the number of hidden neurons
--
ARRAY x[nx] + x0[1] = xx | x0[1] = 1 | --                                     introduce bias
ARRAY v[nv], y[ny], target[ny], error[ny], delta2[nv]
ARRAY WW1[nv, nx + 1], W2[ny, nv], Dww1[nv, nx + 1], Dw2[ny, nv]
--
--                                     random initial weights
for i = 1 to nv
  WW1[i, 1] = 0.2 * ran() | WW1[i, 2] = 0.2 * ran() | W2[1, i] = 0.2 * ran()
next
-----
--                                     set experiment parameters
lrate1 = 1 | lrate2 = 0.3 | mom1 = 0.1 | mom2 = 0.1
scale = 0.5 | NN = 10000
--
for i = 1 to 3 | drun | next | --                                     training runs
lrate1 = 0.4 | lrate2 = 0.15 | --                                     decrease lrate
for i = 1 to 10 | drun | next
--
write "type go for a recall run" | STOP
drun RECALL
-----
DYNAMIC
-----
x[1] = ran() | target[1] = 0.4 * sin(4 * x[1])
invoke NET2(xx, y, v, WW1, W2)
-----
Vector error = target - y
Vector delta2 = W2% * error * (1 - v^2)
MATRIX Dww1 = lrate1 * delta2 * xx + mom1 * Dww1
MATRIX Dw2 = lrate2 * error * v + mom2 * Dw2
DELTA WW1 = Dww1 | DELTA W2 = Dw2
-----
--
label RECALL
x[1] = ran() | target[1] = 0.4 * sin(4 * x[1])
invoke NET2(xx, y, v, WW1, W2)
Vector error = target - y

```

FIGURE 6-4a. Training program and recall test for a two-layer backpropagation network learning the sine function by mean-square regression of a random input on the target function $0.4 * \sin(4 * x[1])$. The network (but in this case not the training program) is defined as a convenient submodel that can be stored and reused with different input, output, and hidden-layer dimensions nx , ny , nv . xx , $WW1$, and $Dww1$ are bias-augmented arrays (Section 6-2).

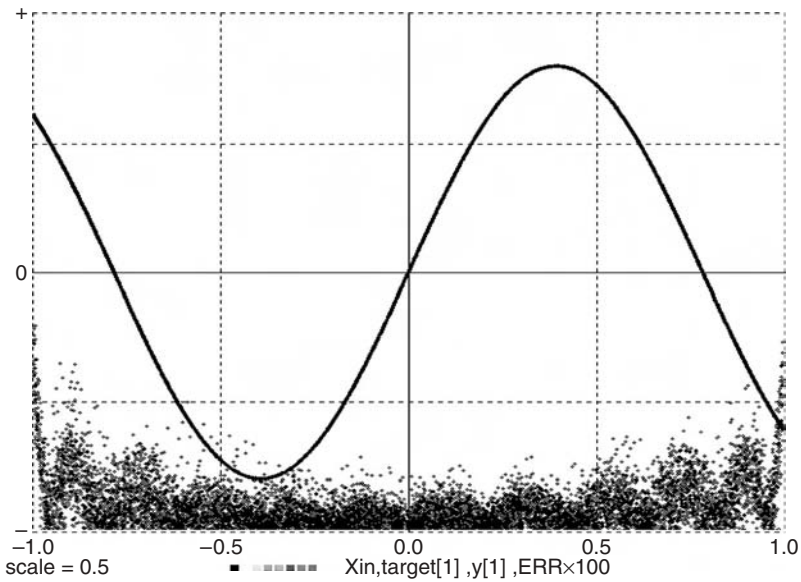


FIGURE 6-4b. Training display produced by the sinusoid-learning program of Figure 6-4a. The network output $y(t)$ and the target sinusoid $\text{target}(t)$ match very accurately, well within the display-curve width. The time history at the bottom represents one hundred times the absolute value of the matching error error .

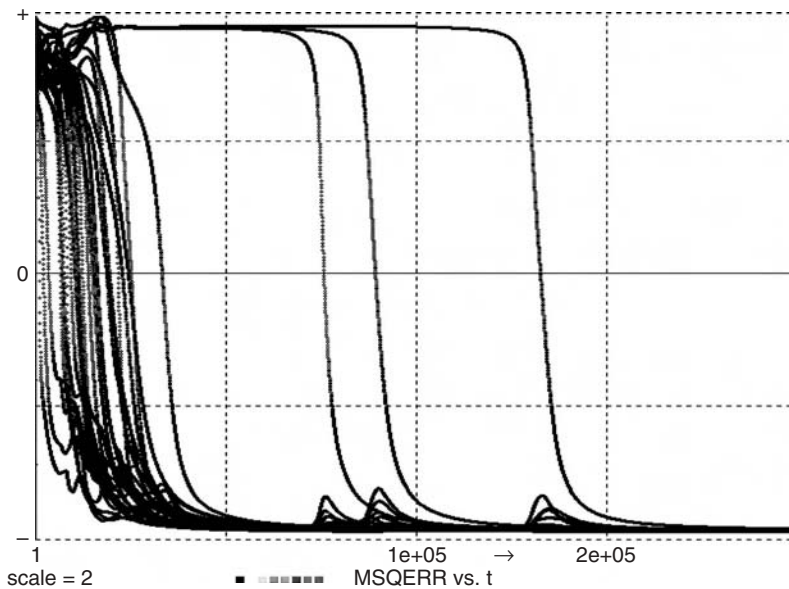


FIGURE 6-4c. Squared-error training histories of 32 pattern-matching errors produced by a larger backpropagation network with one hidden layer and nine hidden neurons. Such optimizations often converge even after temporary instabilities due to excessively large learning rates.

must then find n parameters $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_n$ that minimize the sample average of $\mathbf{g} = [\mathbf{y}(\mathbf{x}) - \mathbf{Y}]^2$. This procedure is readily extended to mean-square regression of n_y -dimensional pattern vectors $\mathbf{y}(\mathbf{x})$ on n_x -dimensional pattern inputs \mathbf{x} (Section 6-6).

We shall approximate the desired output pattern $\mathbf{y} = \mathbf{Y}(\mathbf{x})$ with a single neuron layer that implements

$$y[i] = \sum_{k=1}^n \mathbf{W}[i, k] f_k\{\mathbf{x}[1], \mathbf{x}[2], \dots, \mathbf{x}[n_x]\} \quad (i=1, 2, \dots, n_y) \quad (6-31)$$

with

$$\mathbf{Vector} \mathbf{y} = \mathbf{W} * \mathbf{f} \quad (6-32)$$

where \mathbf{f} is an n -dimensional vector of basis functions $\mathbf{f}[1], \mathbf{f}[2], \dots, \mathbf{f}[n]$. Once these basis functions are computed, we only need to optimize a simple linear network layer. If a minimum exists, successive approximations of the optimal connection weights $\mathbf{W}[i, k]$ are easily computed with the LMS algorithm of Section 6-9, namely,

$$\mathbf{Vector} \mathbf{error} = \mathbf{Y} - \mathbf{y} \quad | \quad \mathbf{DELTA} \mathbf{W} = \text{lr} \mathbf{rate} * \mathbf{error} * \mathbf{f}$$

The matching error **error** is again defined as in Section 6-10.

(b) Radial Basis Functions

Radial-basis-function (RBF) networks employ n hyperspherically symmetrical basis functions $\mathbf{f}[\mathbf{k}]$ of the form

$$\mathbf{f}[\mathbf{k}] = \mathbf{f}(\|\mathbf{x} - \mathbf{X}_k\|; \mathbf{a}[\mathbf{k}], \mathbf{b}[\mathbf{k}], \dots) \quad (k = 1, 2, \dots, n)$$

where the n “radii” $\|\mathbf{x} - \mathbf{X}_k\|$ are the pattern-space distances between the input vector \mathbf{x} and n specified radial-basis centers \mathbf{X}_k in the n_x -dimensional pattern space. $\mathbf{a}[\mathbf{k}], \mathbf{b}[\mathbf{k}], \dots$ are parameters that identify the k th basis function $\mathbf{f}[\mathbf{k}]$. The \mathbf{X}_k and $\mathbf{a}[\mathbf{k}], \mathbf{b}[\mathbf{k}], \dots$ must be judiciously preselected. Truly optimal choices may or may not exist.

The most commonly used radial basis functions are

$$\mathbf{f}[\mathbf{k}] = \exp(-\mathbf{a}[\mathbf{k}]\|\mathbf{x} - \mathbf{X}_k\|^2) \equiv \exp(-\mathbf{a}[\mathbf{k}]\mathbf{rr}[\mathbf{k}]) \quad (k = 1, 2, \dots, n) \quad (6-33)$$

which can be recognized as “Gaussian bumps” for $n_x = 1$ and $n_x = 2$. The radial-basis-function layer is then represented by the simple vector assignment

$$\mathbf{Vector} \mathbf{y} = \mathbf{W} * \exp(-\mathbf{a} * \mathbf{rr}) \quad (6-34)$$

where \mathbf{y} is an n_y -dimensional vector, \mathbf{a} and \mathbf{rr} are n -dimensional vectors, and \mathbf{W} is an $n_y \times n$ connection-weight matrix.

It remains to compute the vector **rr** of squared radii $\mathbf{rr}[k] = \|\mathbf{x} - \mathbf{X}_k\|^2$. Following D.P. Casasent [16] we write the **n** specified radial-basis-center vectors \mathbf{X}_k as the **n** rows of an **n**-by-**nx** pattern-row matrix (template matrix) **P**, that is,

$$(\mathbf{P}[k,1], \mathbf{P}[k,2], \dots, \mathbf{P}[k,nx]) \equiv (\mathbf{X}_k[1], \mathbf{X}_k[2], \dots, \mathbf{X}_k[nx]) \quad (k = 1, 2, \dots, n)$$

(Section 6-5b). Then

$$\mathbf{rr}[k] = \sum_{j=1}^{nx} (\mathbf{x}[j] - \mathbf{P}[k,j])^2 = \sum_{j=1}^{nx} \mathbf{x}^2[j] - 2 \sum_{j=1}^{nx} \mathbf{P}[k,j] \mathbf{x}[j] + \sum_{j=1}^{nx} \mathbf{P}^2[kj] \quad (k = 1, 2, \dots, n)$$

The last term, namely,

$$\sum_{j=1}^{nx} \mathbf{P}^2[kj] = \mathbf{pp}[k] \quad (k = 1, 2, \dots, n)$$

defines an **n**-dimensional vector **pp** that depends only on the given radial basis centers. The DESIRE experiment-protocol script declares and precomputes this constant vector with

```

ARRAY pp[n]
for k = 1 to n
  pp[k] = 0
  for j = 1 to nx | pp[k] = pp[k] + P[k,j]^2 | next
next

```

The DYNAMIC program segment can then generate the desired vector **rr** with

$$\mathbf{DOT\ xx} = \mathbf{x} * \mathbf{x} \quad | \quad \mathbf{Vector\ rr} = \mathbf{xx} - 2 * \mathbf{P} * \mathbf{x} + \mathbf{pp} \quad (6-35)$$

But normally, there is no need to compute **rr** explicitly. From Eq. (6-34), the complete radial-basis-function algorithm is efficiently represented by

$$\begin{aligned} \mathbf{DOT\ xx} &= \mathbf{x} * \mathbf{x} \quad | \quad \mathbf{Vector\ f} = \exp(\mathbf{a} * (\mathbf{2} * \mathbf{P} * \mathbf{x} - \mathbf{xx} - \mathbf{pp})) \\ \mathbf{Vector\ y} &= \mathbf{W} * \mathbf{f} \\ \mathbf{Vector\ error} &= \mathbf{Y} - \mathbf{y} \\ \mathbf{DELTA\ W} &= \mathbf{lr} * \mathbf{error} * \mathbf{f} \end{aligned} \quad (6-36)$$

If desired, one can adjoin a constant bias term to the **f** layer as in Footnote 3.

This combination of Casasent's algorithm and DESIRE vector assignments makes it easy to program RBF networks when we know the number and location of the radial basis centers \mathbf{X}_k and the Gaussian-spread parameters $\mathbf{a}[k]$. But their selection is a real problem, especially when the pattern dimension **nx**

exceeds 2. Tessellation centers produced by competitive vector quantization (Sections 6-15 and 6-16) are often used as radial-basis centers [7,8]. Appendix A shows a complete program.

COMPETITIVE-LAYER PATTERN CLASSIFICATION

6-14. Template-pattern Matching

The classifier networks in Sections 6-7 and 6-10 learned by comparing input patterns with **N** known prototype patterns (supervised learning). A *competitive* pattern classifier learns to associate **nx**-dimensional input patterns $\mathbf{x} \equiv (\mathbf{x}[1], \mathbf{x}[2], \dots, \mathbf{x}[\mathbf{nx}])$ with one of **n** initially unknown patterns (template patterns)

$$\mathbf{W}^{(i)} \equiv (\mathbf{W}[i, 1], \mathbf{W}[i, 2], \dots, \mathbf{W}[i, \mathbf{nx}]) \quad (i = 1, 2, \dots, n \leq \mathbf{nx}) \quad (6-37)$$

so that the sample average of the squared template-matching error

$$\mathbf{g} = \sum_{j=1}^{\mathbf{nx}} (\mathbf{x}[j] - \mathbf{W}[i, j])^2 \quad (i = 1, 2, \dots, n) \quad (6-38)$$

is as small as possible.¹² Such least-squares template matching (*k*-means clustering) partitions the **nx**-dimensional pattern space into **n** Voronoi tessellations (vector quantization). These regions are bounded by hyperplane segments. The statistical relative frequencies of **x** falling into any one tessellation all tend to be approximately equal to **1/n**.

Template-matching classifiers produce **n**-dimensional binary-selector patterns (Section 6-7) corresponding to the **n** template rows. To train a competitive-layer classifier, we feed it successive input patterns **x** and adjust **W** so as to minimize the mean-square template-matching error. The classifier output is then made equal to the binary-selector pattern that identifies the best-matching template. This training process may or may not succeed (Sections 6-16 to 6-18).

¹² Absolute values of $\mathbf{x}[j] - \mathbf{W}[i, j]$ can be used instead and may be easier to compute; squared errors are more tractable mathematically.

6-15. Unsupervised Pattern Classifiers

(a) Simple Competitive Learning

The competitive-classifier layer in Figure 6-5a reads $n \times nx$ -dimensional input patterns \mathbf{x} and computes an $n \times nx$ template matrix \mathbf{W} and an n -dimensional binary-selector output \mathbf{v} . The experiment-protocol script declares

ARRAY $\mathbf{x}[nx]$, $\mathbf{W}[N, nx]$, $\mathbf{v}[n]$

and starts the $\mathbf{W}[i, k]$ with random values (Section 6-10). We supply successive input patterns \mathbf{x} as in Section 6-5. Then every execution of the DYNAMIC-segment statement

CLEARN $\mathbf{v} = \mathbf{W}(\mathbf{x})$ *lrate*, *crit*

with *crit* = -1 finds the template-matrix row ($\mathbf{W}[i, 1], \mathbf{W}[i, 2], \dots, \mathbf{W}[i, nx]$) with the currently smallest squared template-matching error [Eq. (6-38)] and updates this template pattern with

$$\mathbf{W}[i, k] = \mathbf{W}[i, k] + \text{lrate} * (\mathbf{x}[k] - \mathbf{W}[i, k]) \quad (k = 1, 2, \dots) \quad (6-39)$$

(Kohonen–Grossberg learning [15, 22, 24]). The learning rate *lrate* is a positive optimization parameter normally programmed to decrease with successive steps, as in Section 6-9.

The binary-selector output \mathbf{v} in Figure 6-5 identifies the template vector (6-39) closest to the current input \mathbf{x} . To produce this vector for display (Figure 6-6) or further computations, we program

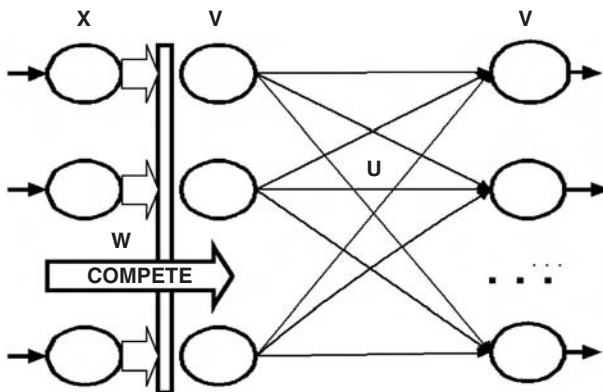


FIGURE 6-5a. A competitive template-matching layer and an optional counterpropagation layer.

DYNAMIC

```

iRow = t | --                select row in pattern matrix INPUT
-- lrate1 = lrate * exp(-SS * t) | -- decrease learn rate (optional)
Vector x = INPUT# + noise * (ran()+ran()+ran()) | -- noisy input
CLEARN y = W(x)lrate1,crit | --                compete,learn
Vectr delta h = y | --                conscience, if any
--
Vector w = W% * y | --    reconstruct and display the templates

```

FIGURE 6-5b. DYNAMIC program segment for a competitive classifier trained with repeated prototype-pattern rows from a pattern-row matrix **INPUT**. Set **crit** = **-1** for simple competitive learning, and **crit** = **0** for FSCL learning (see text). **lrate** = **0** for recall runs. The last line (**Vector w = W% * y**) produces the currently learned template vector **w** corresponding to the input **x**. If these templates successfully approximate the prototype vectors, the classifier functions as an associative memory. To program the counterpropagation layer in Figure 6-5a, we add the lines

```

Vector y = U * v | --                function output
Vector error = target - y | --        output error
DELTA U = lratef * error * v | --    learn function values

```

$$\text{Vector } \mathbf{w} = \mathbf{W\%} * \mathbf{v} \quad (6-40)$$

Ideally, the **n** templates converge to centers of **n** different Voronoi tessellations. This can identify up to **N** noise-perturbed but well separated prototype patterns contained in an input sample (Fig. 6-6a). More often than not, though, a template “following” a prototype pattern **x** in accordance with Eq. (6-39) gets close to a subsequent input pattern and follows it instead. A prototype may then “capture” more than one template vector or none at all, or a template may end up somewhere between prototypes (Section 6-16).¹³ Sections 6-15b and 6-17 describe two schemes for improved competitive learning.

(b) Learning with Conscience

Conscience algorithms [7,8,19,21] bias the template-learning competition so that too-frequently selected templates are given a lower priority. DESIRE can implement the FSCL (frequency-sensitive competitive learning) algorithm of Ahalt et al. [19]. We declare an **n**-dimensional vector **h** $\equiv (\mathbf{h}[1], \mathbf{h}[2], \dots, \mathbf{h}[\mathbf{n}])$ immediately following **v** with

ARRAY ... , v[N], h[n], ...

¹³ In effect, the process has converged to a local minimum of the mean-square template-matching error rather than to its global minimum.

and program

$$\text{Vector } \mathbf{h} = \mathbf{h} + \mathbf{v} \quad \text{or} \quad \text{Vectr delta } \mathbf{h} = \mathbf{v} \quad (6-41)$$

in a DYNAMIC program segment. Each $\mathbf{h}[i]$ then starts at 0 and counts the number of times the i th template was selected in the course of training. Then

CLEARN $\mathbf{v} = \mathbf{W}(\mathbf{x})$ *lrate*, *crit*

with **crit** = 0 finds and updates the template row ($\mathbf{W}[1, 1], \mathbf{W}[1, 2], \dots, \mathbf{W}[1, n_x]$) with the smallest product of the count $\mathbf{h}[i]$ and the current squared template-matching error [Eq. (6-38)]. This tends to equalize the template-matching counts and improves competitive learning. The results are still not always perfect, as shown in the following section.

6-16. Experiments with Pattern Classification and Vector Quantization

The program in Figure 6-5 permits a wide range of experiments.

(a) Pattern Classification

Figure 6-6a illustrates classification of noise-perturbed two-dimensional prototype pattern vectors represented as points $\mathbf{s} \equiv (\mathbf{s}[1], \mathbf{s}[2])$. The experiment protocol generates \mathbf{N} simple two-dimensional prototype patterns (\mathbf{N} uniformly spaced points in a square) and stores them as rows of the $\mathbf{N} \times 2$ pattern-row matrix **INPUT** (Section 6-5b). A DYNAMIC program segment adds approximately Gaussian noise to produce the classifier input \mathbf{x} with

$$\text{Vector } \mathbf{x} = \text{INPUT\#} + \text{noise} * (\text{ran}() + \text{ran}() + \text{ran}()) \quad (6-42)$$

We set **iRow** = **t** to present the \mathbf{N} prototypes in sequence (a random sequence yields similar results). Alternatively, **iRow** = **t/m** presents each pattern **m** times to let the classifier learn each pattern in turn. The DYNAMIC-segment statement

CLEARN $\mathbf{v} = \mathbf{W}(\mathbf{x})$ *lrate*, *crit*

with **crit** set to 0, -1, or a positive value allows trying different types of classifiers (Sections 6-16 and 6-17). One can also vary **N**, **n**, *lrate*, and the noise level **noise**.

Clearly, classifying \mathbf{N} prototype patterns requires at least **n** templates. But training with a repeated or random sequence of different prototypes is likely

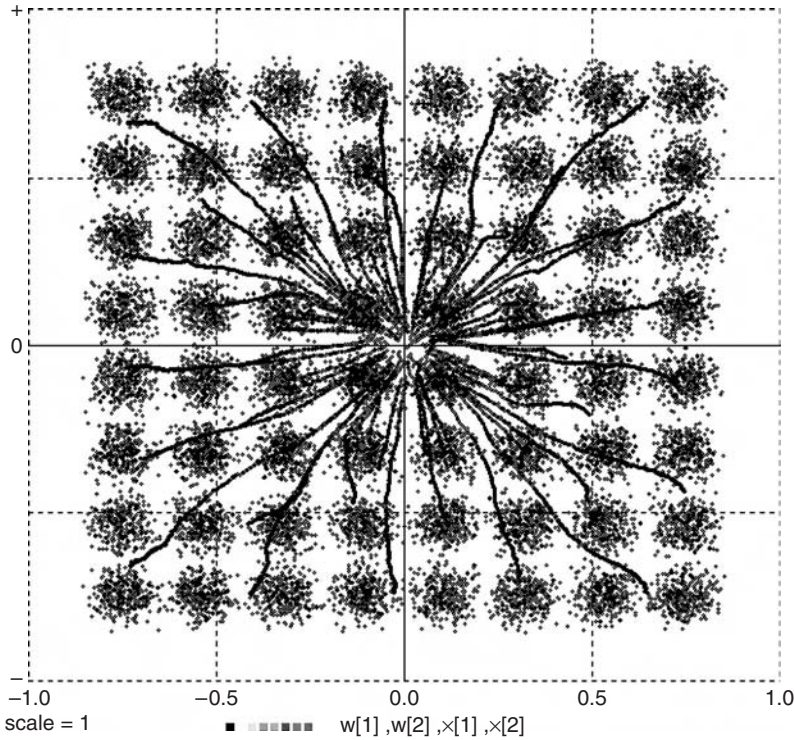


FIGURE 6-6a. Competitive learning of two-dimensional patterns. Referring to Figure 6-5b, the display shows $N = 64$ noisy pattern inputs $\mathbf{x} \equiv (\mathbf{x}[1], \mathbf{x}[2])$ and some computed template vectors $\mathbf{w} \equiv (\mathbf{w}[1], \mathbf{w}[2])$ trying to approximate the 64 prototype patterns. With simple competitive learning, some templates may end up between input-pattern clusters, and some clusters may attract more than one template vector.

to fail unless n is larger (possibly substantially larger) than N . In that case, two or more different binary-selector outputs correctly identify the same prototype.

(b) Vector Quantization

When the prototype-pattern input (6-42) is replaced in the competitive-layer program of Figure 6-5b with a pure noise input, say,

$$\mathbf{x} = \text{ran}()$$

the template updating (6-39) tends to move the n computed template vectors \mathbf{w} to the centers of n Voronoi tessellations in the $n\mathbf{x}$ -dimensional input-pattern space (Fig. 6-6b). The binary selector output \mathbf{v} identifies the

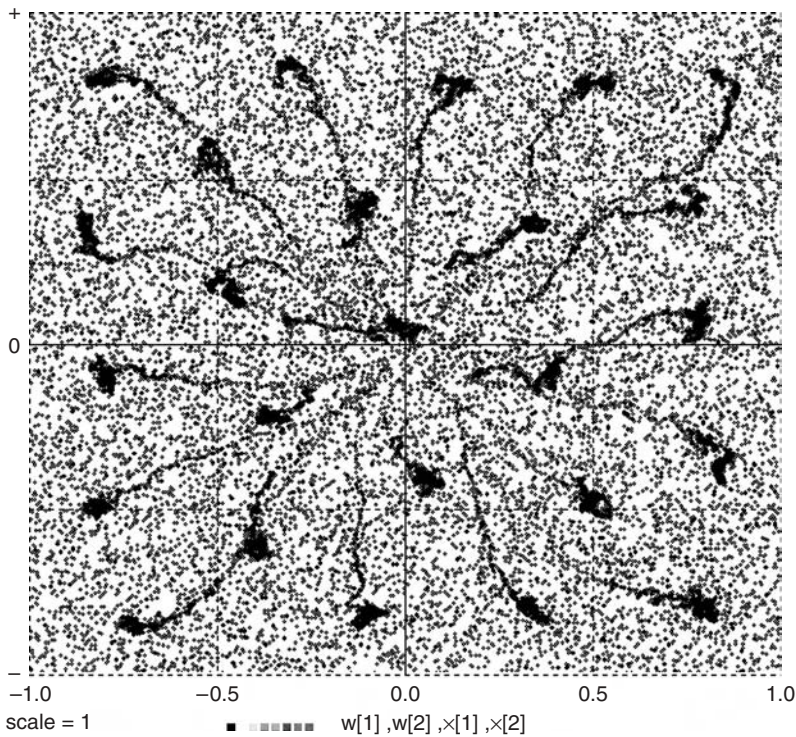


FIGURE 6-6b. Here, $N = 15$ template vectors \mathbf{w} are trying to learn Voronoi-tessellation centers for a pure-noise input $\mathbf{x} = \text{ran}()$ uniformly distributed over a square. Results are not perfect even with conscience-assisted learning ($\text{crit} = 0$). The Appendix shows an application.

tessellation region that matches \mathbf{x} best. For $\text{crit} = 0$ (FSCL learning, Section 6-15b), $\mathbf{h}[\mathbf{i}]/t$ estimates the statistical relative frequency of finding \mathbf{x} in the i th tessellation in t trials; all $\mathbf{h}[\mathbf{i}]$ ought to approach t/n as t increases. Actual experiments confirm these theoretical predictions only approximately.

6-17. Simplified Adaptive-resonance Emulation

The Carpenter–Grossberg adaptive resonance theory (ART) [22–26] deals with the multiple-capture problem by updating an already-committed template only if it matches the current input within a preset vigilance limit (“resonance”). Otherwise, a reset operation eliminates the template from the competition, which then selects the next-best template, possibly a new pseudorandom template. ART preserves already-learned pattern categories.

The DYNAMIC-segment operation¹⁴

$$\text{CLEARN } \mathbf{v} = \mathbf{W}(\mathbf{x}) \text{ lrate, crit} \quad (6-43)$$

with **crit** > 0 implements ART functionality for the common special case of in-turn pattern learning with low noise [28]. The program is outlined in the Appendix. This classifier will not let successive prototypes “steal” committed templates and learns $\mathbf{n} = \mathbf{N}$ noise-free prototype patterns flawlessly one after another ($\mathbf{iRow} = \mathbf{t/m}$). The algorithm also tolerates a small amount of additive noise (Fig. 6-7*b*); with too much noise, the process runs out of templates and returns an error message. As in classical ART [24], a fast-learn mode for noise-free patterns simply sets $\mathbf{W}^{(l)} = \mathbf{x}$ instead of gradual updating when you replace **crit** in statement (6-43) with **crit #**.

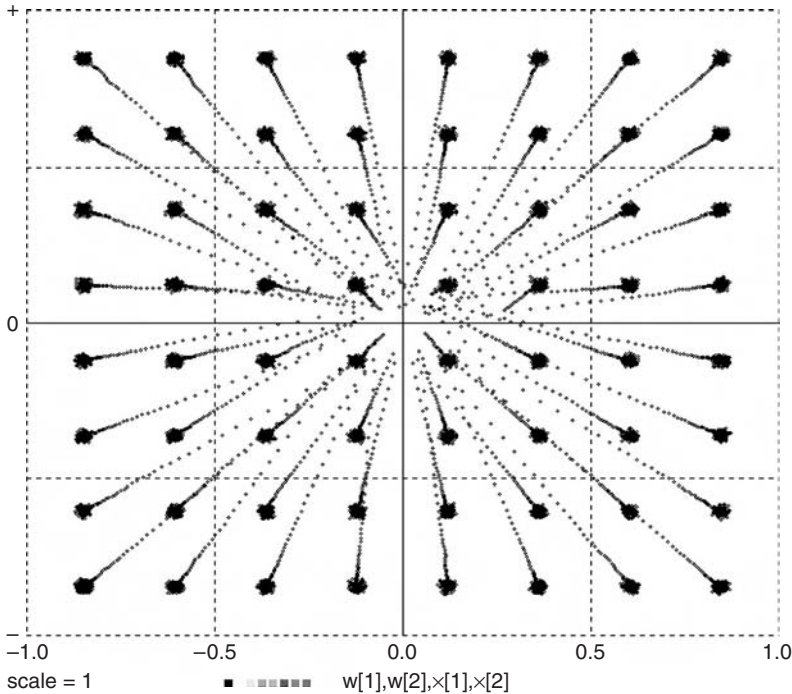


FIGURE 6-7a. Competitive template matching of $\mathbf{N} = 64$ noisy patterns $\mathbf{x} \equiv (\mathbf{x}[1], \mathbf{x}[2])$ with the pseudo-adaptive resonance scheme of Section 6-17 (**crit** = 0.015). Note that pattern clusters do not “steal” each other’s templates. Repeated presentation of the 64 noise-corrupted prototype patterns in turn ($\mathbf{iRow} = \mathbf{t/m}$ with $\mathbf{m} = 200$) produced flawless template matching, but only with low noise levels. The original display was in color.

¹⁴ Note that the DESIRE **CLEARN** operation is different from the **CLEARN** operation used with the early version of DESIRE described in Reference [17]. (See also Section A-3.)

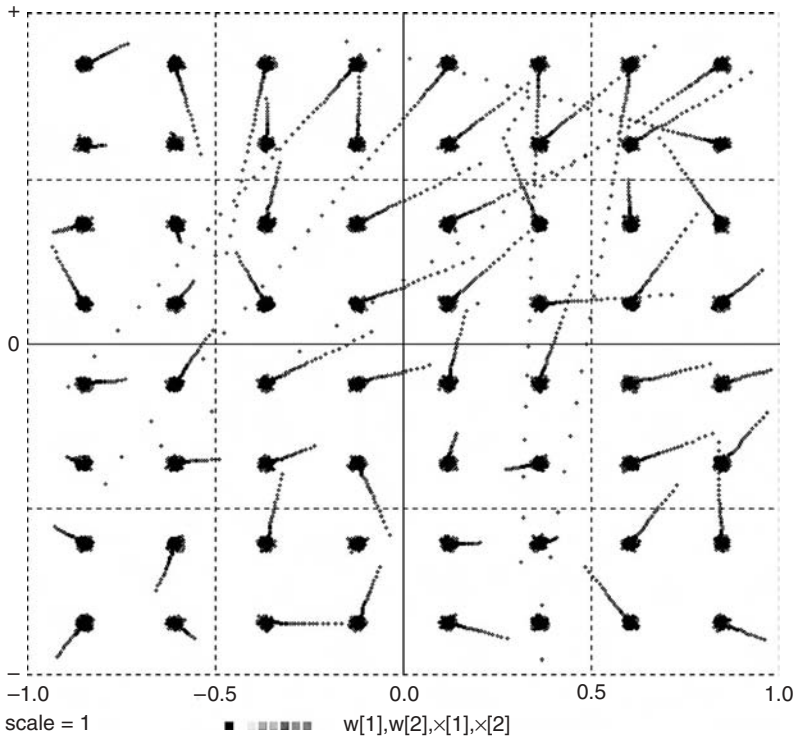


FIGURE 6-7b. The same experiment with larger initial random template-vector values. We had to increase the number n of available templates from 64 to 70 to match all $N = 64$ patterns.

6-18. Biologically Plausible Competition: Correlation Matching

The abstract **CLEARN** routines in Sections 6-16a, b, and 6-18 (see also the **DESIRE** Manual in the book CD) simplify programming, but we may want to see how biologically more plausible neural networks might model pattern-matching competition. Specifically, such models ought to relate pattern matching and contrast enhancement to connection weights that represent physical synapses. Newer biological models use pulsed-neuron replication (Sections 6-24 and 6-25) [29,31].

For Euclidean-normalized patterns \mathbf{x} (Section 6-3), minimizing the sample average of the squared template-matching error (6-38) is equivalent to correlation matching, which maximizes the sample average of

$$g1 = \sum_{j=1}^{nx} W[i, j]x[j] \quad (6-44)$$

A contrast-enhancing neural-network layer (Section 6-3) represented by

$$\text{Vector } \mathbf{v}^{\wedge} = \mathbf{W} * \mathbf{x} \quad | \quad \text{Vector } \mathbf{v} = \text{switch}(\mathbf{v})$$

not only computes the correlation function (6-44) but also generates binary-selector patterns \mathbf{v} that identify the best-matching template-row, that is, the template with the largest correlation function. What is more, the template-vector components to be adjusted now appear as synapse-modeling neuron connection weights.

The template-updating operations (6-39) can be written as a matrix difference equation (Section 3-10), and the entire correlation-matching operation is represented by¹⁵

$$\begin{array}{lll} \text{DOT } \mathbf{xnormsq} = \mathbf{x} * \mathbf{x} & | & \mathbf{xnn} = 1/\text{sqrt}(\mathbf{xnormsq}) \quad | \quad \text{Vector } \mathbf{x} = \mathbf{xnn} * \mathbf{x} \\ \text{Vector } \mathbf{v}^{\wedge} = \mathbf{W} * \mathbf{x} & | & \text{Vector } \mathbf{v} = \text{switch}(\mathbf{v}) \\ \text{Vector } \mathbf{w} = \mathbf{W} \% * \mathbf{v} & | \text{--} & \text{reconstruct templates} \\ \mathbf{e} = \mathbf{x} - \mathbf{w} & \text{--} & \text{template-matching error} \\ \text{DELTA } \mathbf{W} = \text{irate} * \mathbf{v} * \mathbf{e} & | \text{--} & \text{update} \end{array}$$

Note that we implemented the normalization and contrast-enhancing layers with simple assignments, as in Section 6-3. Biologically even more plausible models learn these features gradually as training proceeds, using special nonlinear neurons for normalization and lateral feedback between adjacent neurons for contrast enhancement [15, 17, 20, 22]. Such programs can be made to work, but they are complicated especially when more than a few neural-network layers are needed. Abstract operations such as **CLEARN** $\mathbf{v} = \mathbf{W}(\mathbf{x})$ **irate**, **crit** are easier to program.

SUPERVISED COMPETITIVE LEARNING

Competitive-layer classifiers also work with supervised training for pattern classification, regression, and associative memory. Such neural networks may converge more easily than backpropagation networks.

6-19. Supervised Competitive Classifiers: The LVQ Algorithm

Kohonen's LVQ (learning vector quantization) algorithm [8,15] modifies the competitive-classifier updating rule (6-39) for supervised competitive learning. Each training-sample input \mathbf{x} is presented together with its known associated binary-selector pattern \mathbf{S} , as in Section 6-10. If the competitive-layer

¹⁵ Example **nquant.lst** in the book CD shows a complete program that also computes statistical relative frequencies for each template pattern.

selector output \mathbf{v} does not match \mathbf{S} , the sign of the learning rate \mathbf{lrate} in Eq. (6-39) is reversed.

6-20. Counterpropagation Networks

Hecht-Nielsen's counterpropagation network (Fig. 6-5; [21]) feeds the binary-selector output \mathbf{v} of a competitive-layer classifier to an "outstar" layer programmed with

$$\text{Vector } \mathbf{y} = \mathbf{U} * \mathbf{v}$$

\mathbf{U} may be known from a separate computation, or it can be LMS-trained to associate a desired output vector $\mathbf{y} = \text{target}$ with each pattern input \mathbf{x} . \mathbf{U} can be trained during or after the competitive learning process.

Counterpropagation networks usually approximate regression more quickly than backpropagation networks. But the resulting approximation \mathbf{y} is not continuous: \mathbf{y} can take only \mathbf{n} different values, where \mathbf{n} is the chosen number of template vectors. These values will be spaced most closely in regions corresponding to frequent inputs (Fig. 6-8). Radial-basis-function networks (Section 6-13b) with competitive basis-center learning are, in effect, counterpropagation networks with built-in interpolation. Reference [17] shows some examples.

NEURAL NETWORKS WITH MEMORY

6-21. Neural Networks and Memory

With all connection-weight values set, the neural networks programmed in Sections 6-1 to 6-20 are static function generators. A time series $\mathbf{x} = \mathbf{x}(\mathbf{t})$ of input patterns produces corresponding output patterns $\mathbf{y} = \mathbf{y}(\mathbf{t})$ without memory of past inputs or outputs. Neural-network training or adaptation to changing patterns, though, implies memory, as the network output feeds back to the connection weights. Such adaptation is normally slow relative to input-signal changes (low values of \mathbf{lrate} for stable learning, Section 6-9), and this type of neural-network memory is referred to as long-term memory.

Short-term memory converts some neuron activations into state variables related to past values of inputs or other neuron activations. Such neural networks are not static function generators; they are dynamic systems (filters). Their connection weights can learn to recognize pattern sequences, or learn to

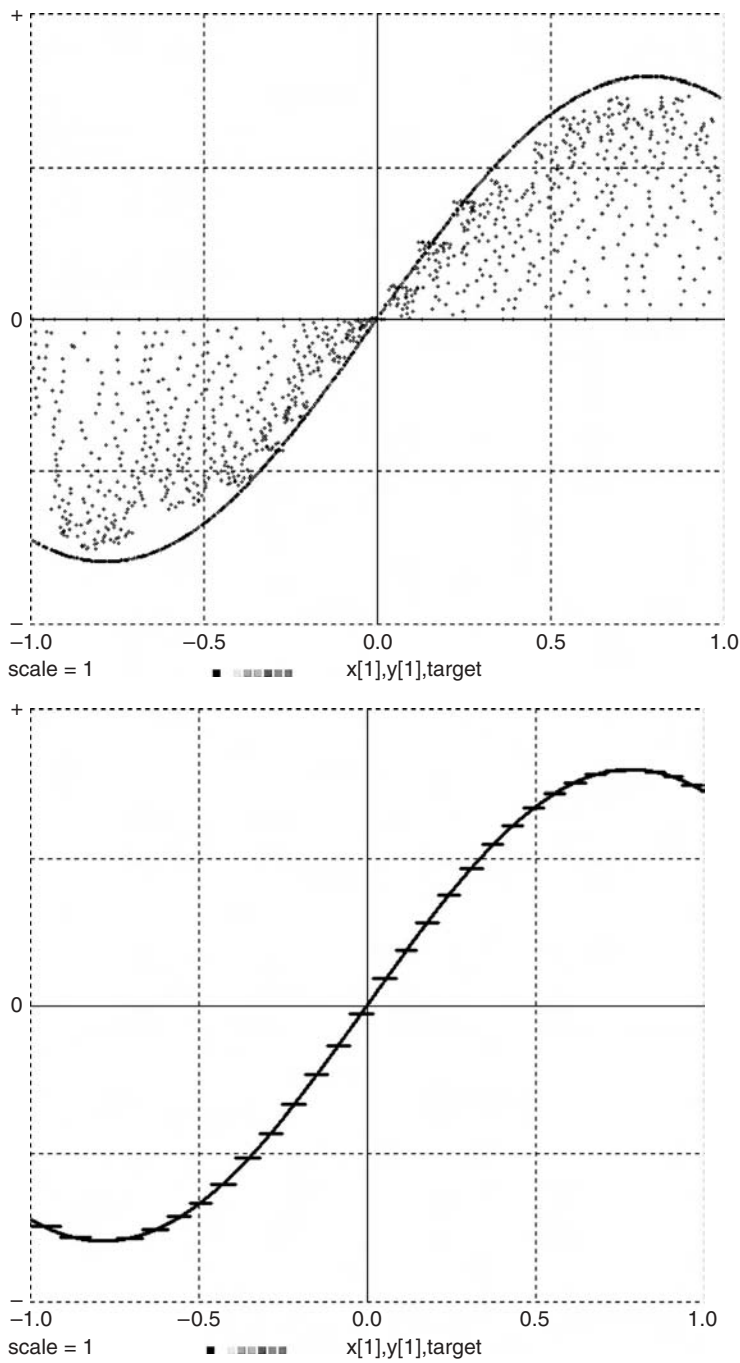


FIGURE 6-8. Counterpropagation learning of a sine-wave input. Note that the output is a step function.

match the outputs of a dynamic (state-equation) model fed a given sample of input signals (model matching, a generalization of regression).

Short-term memory takes different forms, which can also be combined:

- tapped delay lines (shift registers) feeding neuron layers or individual neurons
- neurons modeled by difference-equation systems or differential-equation systems
- feedback to preceding neuron layers (recurrent neural networks)

Much of this large subject is still in the research stage [7,14,36]. As before, we present no exhaustive treatment but point out examples where our compact vector models can be helpful.

6-22. Networks with a Delay-line Input Layer

(a) Vector Model of a Tapped Delay Line

In Figure 6-9, a static neural network reads the activations $\mathbf{x}[1]$, $\mathbf{x}[2]$, ..., $\mathbf{x}[\mathbf{nx}]$ of a tapped-delay-line layer fed with a scalar function $\mathbf{s}(\mathbf{t})$ of the trial number \mathbf{t} . To implement the \mathbf{nx} -stage delay line, we declare an \mathbf{nx} -dimensional vector \mathbf{x} in the experiment protocol and program the DYNAMIC-segment assignments

$$\text{Vector } \mathbf{x} = \mathbf{x}\{-1\} \quad | \quad \mathbf{x}[1] = \mathbf{s}(\mathbf{t}) \quad (6-45)$$

The order of these two assignments is significant. The current value of the input signal $\mathbf{s}(\mathbf{t})$ is read after the index-shift operation (Section 3-6) has

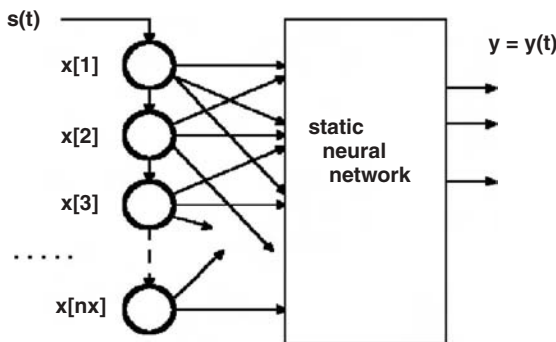


FIGURE 6-9. A delay-line layer feeding a static neural network. Either a simple tapped delay line or a gamma delay line can be used.


```

NN=5
ARRAY x[4]
drun
-----
DYNAMIC
-----
Vector x = x{-1} | x[1] = t
type x[1], x[2], x[3], x[4]

      t           x[1]           x[2]           x[3]           x[4]
1.00000e+00  1.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
2.00000e+00  2.00000e+00  1.00000e+00  0.00000e+00  0.00000e+00
3.00000e+00  3.00000e+00  2.00000e+00  1.00000e+00  0.00000e+00
4.00000e+00  4.00000e+00  3.00000e+00  2.00000e+00  1.00000e+00
5.00000e+00  5.00000e+00  4.00000e+00  3.00000e+00  2.00000e+00

```

FIGURE 6-10. This small program illustrates the operation of a tapped delay line by listing $x[1]$, $x[2]$, ... at successive sampling times $t = 1, 2, \dots$. Note that each new value of the input is read after the line shifts.

shifted earlier samples of $s(t)$ into successive delay-line neurons,¹⁶ so that (Fig. 6-10)

$$x[1] = s(t), x[2] = s(t - \text{COMINT}), x[3] = s(t - 2 \text{ COMINT}), \dots$$

As noted in Section 6-5, $\text{COMINT} = \text{TMAX}/(\text{NN} - 1)$ automatically defaults to 1 and $t = 1, 2, \dots$ if t_0 and TMAX are not specified.

(b) Simple Linear Filters

In Figure 6-9, the neural network connected to the delay line is a static neural network and can be trained as in the preceding sections. The simplest such network just computes a weighted sum

$$\begin{aligned}
 y(t) &= w_1 x[1] + w_2 x[2] + \dots + w_n x[n] \\
 &\equiv w_1 s(t - \text{COMINT}) + w_2 s(t - 2 \text{ COMINT}) + \dots \\
 &\quad + w_n s(t - n \text{ COMINT})
 \end{aligned} \tag{6-46a}$$

with

$$\text{DOT } y = w * x \tag{6-46b}$$

¹⁶ Note that shifting in the opposite direction would require an auxiliary vector to avoid illegal recursion (Section 3-6) [17].

This creates a linear finite-impulse response (FIR) digital filter defined by the connection-weight vector $\mathbf{w} \equiv (\mathbf{w1}, \mathbf{w2}, \dots)$. The design of such filters is discussed in References [7] and [30]. Widrow's LMS algorithm (Section 6-9) was, in fact, first developed to optimize such filters [7].

(c) Linear Matched Filters, Signal Classifiers, and Model Matching

More generally, we can program an \mathbf{ny} -dimensional linear neuron layer

$$\text{Vector } \mathbf{y} = \mathbf{W} * \mathbf{x} \quad (6-47)$$

to implement \mathbf{ny} linear filters such as matching filters for \mathbf{ny} different input signals $\mathbf{s(t)}$ [7]. We can also use the softmax-classifier layer described in Section 6-10 to classify temporal signal patterns shifted into our tapped delay line [7]. Last, but not least, note that the single DESIRE program line

Vector $\mathbf{x} = \mathbf{x}\{-1\}$ | $\mathbf{x}[1] = \mathbf{s(t)}$ | Vector $\mathbf{y} = \mathbf{W} * \mathbf{x}$

combines a tapped delay line and a linear neuron layer into a general linear-system model; the connection weights $\mathbf{W[i, k]}$ can even be given functions of the time \mathbf{t} . Such models can be used to approximate the input/output behavior of real systems such as process plants [7].

(d) A Nonlinear Predictor Trained with Backpropagation

Linear filters have long been used for prediction, but a nonlinear neural network can do better [7]. The program of Figure 6-11 feeds the delay-line activations $\mathbf{x[k]}$ to the hidden layer of a simple two-layer network (Section 6-12)

Vector $\mathbf{vv} = \tanh(\mathbf{WW1} * \mathbf{xx})$

Vector $\mathbf{y} = \mathbf{WW} * \mathbf{vv}$

\mathbf{xx} and \mathbf{vv} are bias-augmented vectors declared with

ARRAY $\mathbf{x[nx]} + \mathbf{x0[1]} = \mathbf{xx}$ | $\mathbf{x0[1]} = 1$ | -- delay line
ARRAY $\mathbf{v[nv]} + \mathbf{v0[1]} = \mathbf{vv}$ | $\mathbf{v0[1]} = 1$ | -- hidden layer

as shown in Footnote 3.

The output vector \mathbf{y} is one-dimensional, and $\mathbf{S(t)} = \mathbf{y[1]}$ is the predictor output signal. Assuming that the statistical properties of our signals do not change with time, the program in Figure 6-11 generates training samples of the “future” signal $\mathbf{sTRUE(t)}$ and delays \mathbf{sTRUE} by the prediction time \mathbf{m} **COMINT = m** with another delay line

Vector $\mathbf{signal} = \mathbf{signal}\{-1\}$ | $\mathbf{signal[1]} = \mathbf{sTRUE}$

```

--          NONLINEAR ADAPTIVE PREDICTOR
--          predicts Mackay-Glass chaos generator
-----

display N1 | display C8 | scale = 3
irule 1 | -- Euler integration for McKay-Glass
TMAX = 500 | DT = 0.04 | NN = TMAX/DT
a = 0.2 | b = 0.1 | c = 10 | -- for Mackay-Glass
tau = 25

--
ARRAY DD[1000] | -- time-delay buffer
sTRUE = 10 | - initialize time-delay buffer
for i = 1 to 1000 | DD[i] = sTRUE | next
-----

m = 20 | -- predictor delay
nx = 50 | -- number of predictor neurons
nv = 13 | -- number of hidden-layer neurons
--
ARRAY signal[m], y[1], error[1]
ARRAY x[nx] + x0[1] = xx | x0[1] = 1 | -- delay line
ARRAY v[nv] + v0[1] = vv | v0[1] = 1 | -- hidden layer
ARRAY vdelta[nv] + v0delta[1] = vvdelta | v0delta[1] = 1
ARRAY WW1[nv+1, nx+1], WW2[1, nv+1]
--
for i = 1 to nv+1 | for k = 1 to nx+1 | -- initialize
    WW1[i, k] = 0.1 * ran()
next | next
-----

WW1gain = 0.06 | WW2gain = 0.006
--
N = 20
for i = 1 to N | drun | next | -- N training runs
write 'type go for prediction tests' | STOP
WW1gain = 0 | WW2gain = 0 | drun | drun | -- test runs
-----

DYNAMIC
-----

tdelay Sd = DD, sTRUE, tau | -- McKay-Glass time series
sTRUEdot = a * Sd/(1 + Sd^c) - b * sTRUE
d/dt sTRUE = sTRUEdot
-----

-- delay "future" signal sTRUE, shift resulting signal into x
--
-- OUT is not needed with Euler integration

```

FIGURE 6-11. Complete program for the nonlinear predictor.

```

Vector signal = signal{-1} | signal[1] = sTRUE
Vector x = x{-1} | x[1] = signal[m]
-----
Vector vv = tanh(WW1 * xx) | -- note bias
Vector y = WW2 * vv | -- no limiter needed on output!
--
Vector error = sTRUE - y | -- backpropagation
Vector vvdelta = WW2% * error * (1 - vv^2)
DELTA WW1 = WW1gain * vvdelta * xx
DELTA WW2 = WW2gain * error * vv
-----
ERRORx5 = 5 * error[1] - 0.5 * scale
dispt y[1], ERRORx5, sTRUE

```

FIGURE 6-11. (Continued).

to produce the current predictor input $\mathbf{s}(t) = \mathbf{signal}[m]$. The simulated predictor then tries to predict $\mathbf{sTRUE}(t)$ by minimizing the sample average of $\mathbf{g} = (\mathbf{y} - \mathbf{sTRUE})^2$ with the backpropagation algorithm of Section 6-12a. Prediction results necessarily depend on the frequency content of the input signal.

To provide a fairly difficult prediction task, $\mathbf{sTRUE} = \mathbf{sTRUE}(t)$ is the Mackay–Glass “chaotic” time series [17] defined by¹⁷

$$\begin{aligned} \mathbf{Sd}(t) &= \mathbf{sTRUE}(t - \tau) \\ (d/dt) \mathbf{sTRUE} &= a \mathbf{Sd}/(1 + \mathbf{Sd}^c) - b \mathbf{sTRUE} \end{aligned}$$

Figure 6-12 shows the “future” signal value \mathbf{sTRUE} , the predictor output \mathbf{y} , and the prediction error $\mathbf{sTRUE} - \mathbf{y}$ during training and recall. We used 20 training runs with a total of 250,000 training steps to learn prediction $\mathbf{m} = 20$ steps ahead.

¹⁷ In Figure 6-11, this is programmed with

```

tdelay Sd = DD, sTRUE, tau
sTRUEdot = a * Sd/(1 + Sd^c) - b * sTRUE
d/dt sTRUE = sTRUEdot

```

where **tdelay** is a library time-delay routine that implements $\mathbf{Sd}(t) = \mathbf{sTRUE}(t - \tau)$ by storing samples of its input \mathbf{sTRUE} in an array **DD** declared in the experiment protocol; one sample for each **DT** step of the simple Euler integration routine (Section 1-7a) used here. The example **mglass.lst** in the book CD lets you experiment with the generator.

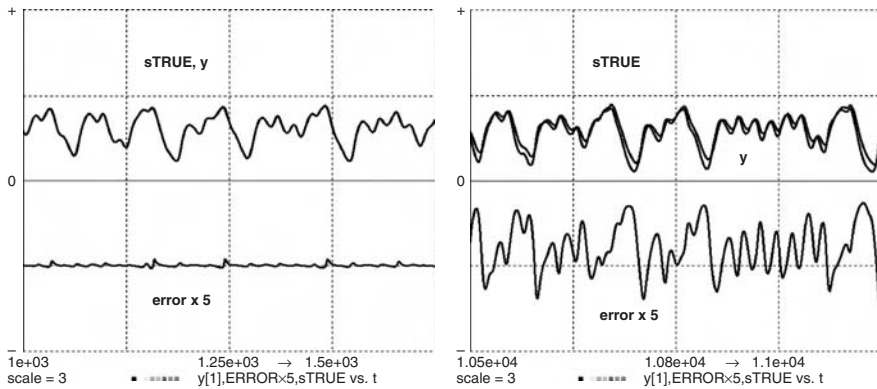


FIGURE 6-12. Time histories of the “future” signal **sTRUE**, the predictor output **y**, and the scaled predictor error **5(sTRUE – y)** during training and recall.

6-23. The Gamma Delay Line Layer

J. Principe and his associates [7] replaced the tapped-delay-line definition (6-45), or

$$\mathbf{x}[i] = \mathbf{x}[i - 1] \quad (i = 2, 3, \dots, \mathbf{nx}) \quad \mathbf{x}[1] = \mathbf{s}(t) \quad (6-48)$$

with a cascade of simple difference equations

$$\mathbf{x}[i] = \mathbf{x}[i] + \mu(\mathbf{x}[i - 1] - \mathbf{x}[i]) \quad (i = 2, 3, \dots, \mathbf{nx}) \quad \mathbf{x}[1] = \mathbf{s}(t) \quad (6-49a)$$

where **μ** is a positive parameter. Our compact vector notation models this gamma delay line with a single vector difference equation (Section 3-4)

$$\text{Vectr delta } \mathbf{x} = \mu * (\mathbf{x}\{-1\} - \mathbf{x}) \quad | \quad \mathbf{x}[1] = \mathbf{s}(t) \quad (6-49b)$$

Each tapped-delay-line neuron [Eq. (6-46)] “remembers” just one past input value. But each neuron output **x[i]** in the gamma delay line is affected by all past input values. This extra information about the past history of **s(t)** may allow a reduction in the number **nx** of delay-line sections in the block diagram of Figure 6-9 compared to that needed with a simple tapped delay line.

Figure 6-13 displays the tap-value responses to the initial condition **x[1] = 1** for **nx = 8** and two different values of **μ**. The memory effect decreases with time. The maximum time interval analyzed by a delay-line-fed neural network is **nx COMINT** for a simple tapped delay line. For a gamma delay line, the effective memory period (memory depth) still depends on **nx** but is mainly determined by the difference-equation parameter **μ**. Suitable values

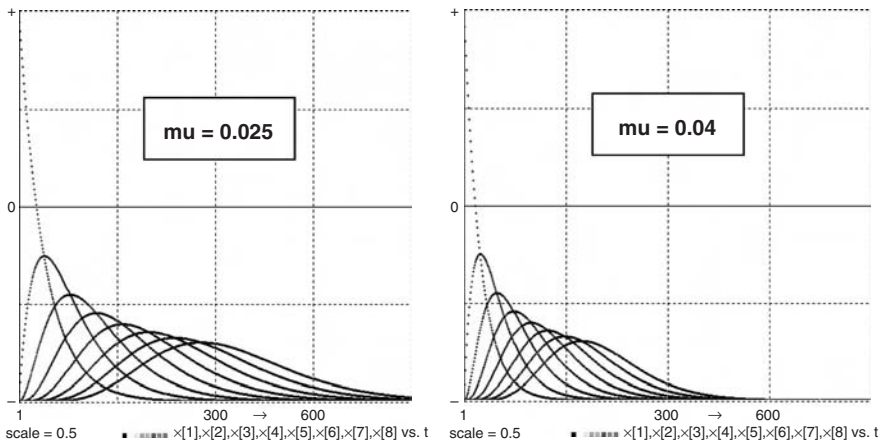


FIGURE 6-13. Response of the tap outputs of an 8-tap gamma delay line to the initial condition $x[1] = 1$ for $\mu = 0.025$ and $\mu = 0.04$. Curves in the original display were in different colors; the small squares at the bottom are color keys.

of this parameter are often found by trial and error; References [7] and [14] discuss automatic training.

The simplest static neural networks used with an input gamma delay line are again linear (weighted-sum) layers (6-46) or (6-47), which can be optimized with the LMS algorithm. The tap activation functions in Figure 6-13 serve as a useful set of basis functions for regression, as in Eq. (6-31). Reference [7] discusses more advanced networks and a number of applications.

PULSED-NEURON REPLICATION

6-24. Pulsed-neuron Models

Biological neurons propagate electrical signals, but their actual inputs and outputs are fluctuating release rates of chemical substances (transmitters) fed into synaptic clefts between neurons [17, 23]. Neuron activations in the simplified neural networks discussed in Sections 6-1 to 6-23 model running averages of pulsed-neuron inputs and outputs.

In the receiving neuron, a transmitter substance reacts with receptor chemicals to change the neuron-membrane permeability to ions passing into and out of the neuron. Multiple excitatory and/or inhibitory inputs roughly add with different individual “gains” and “fire” the neuron when their weighted sum exceeds a threshold value. “Firing” or ion transition through the neuron membrane produces a positive 20 to 300-mV voltage pulse across the membrane at a specific location. This pulse propagates down a neuron fiber (axon)

as the local positive voltage makes successive cell-membrane locations more permeable to sodium ions from the outside. The propagated pulse eventually produces chemical output by releasing transmitter substance at an output synapse. The membrane voltage subsides as positive potassium ions leave the cell, and the cell relaxes to restore equilibrium ion densities.

The model in Figure 6-14*a* has an integrate-and-fire block producing pulses $y(t)$ shaped like those in Figures 6-14*b* and *c*, and one or more delay sections modeling pulse propagation toward the output synapse. The pulse generator integrates a positive input x until the output y reaches a threshold value, and then produces a finite-duration pulse. $y(t)$ returns to its initial value in the course of a refractory period determined by ion-motion delays. If the positive input persists, the process repeats and generates a pulse train. The pulse frequency increases with the input amplitude but is limited to about 1 kHz by the refractory periods.

The simplest delay sections are represented by differential equations

$$(d/dt) q_{out} = q_{in} - q_{out}/\tau$$

whose time constants τ depend on ion exchanges and myelin-sheath insulation in successive membrane sections. More complicated differential-equation systems can be substituted.

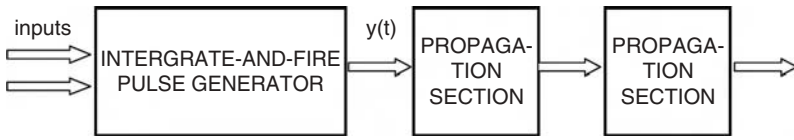
Each pulsed neuron, then, is represented by a few differential equations that involve switching functions. Once one decides on such a neuron model, it is easy to simulate layers and groups of neurons by replacing differential equations and defined-variable assignments with DESIRE vector differential equations and vector assignments (Section 6-25). Multiple neuron inputs can be readily modeled by replacing the input x with a matrix-vector product $A * x$ or with $\text{sigmoid}(A * x)$. The resulting pulsed-neuron models can be used in networks such as those in the preceding sections, or in entirely new combinations, especially for biological modeling.

6-25. A Simple Integrate and Fire Model

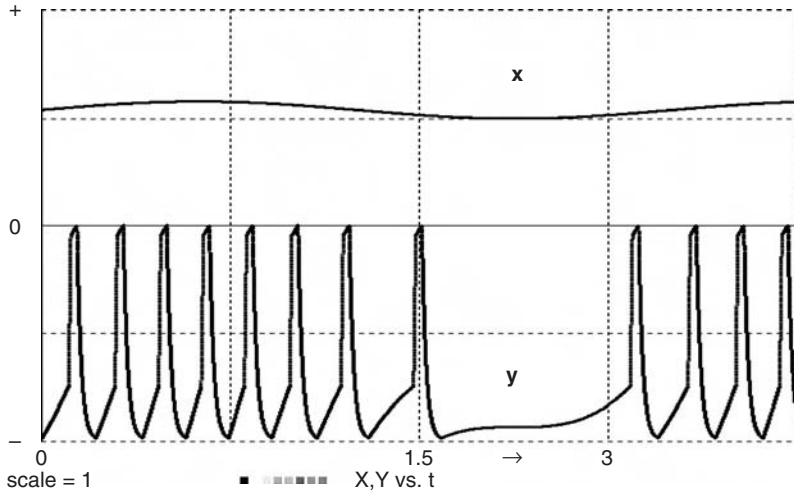
In the DYNAMIC-segment program of Figure 6-15, the neuron input x can be a weighted sum of excitatory positive inputs and inhibitory negative inputs; only positive values of x can fire the neuron. A neuron pulse $y(t)$ with positive rest value y_0 and peak value $\text{peak} > y_0$ is produced by an integrator modeled with

$$d/dt y = ydot$$

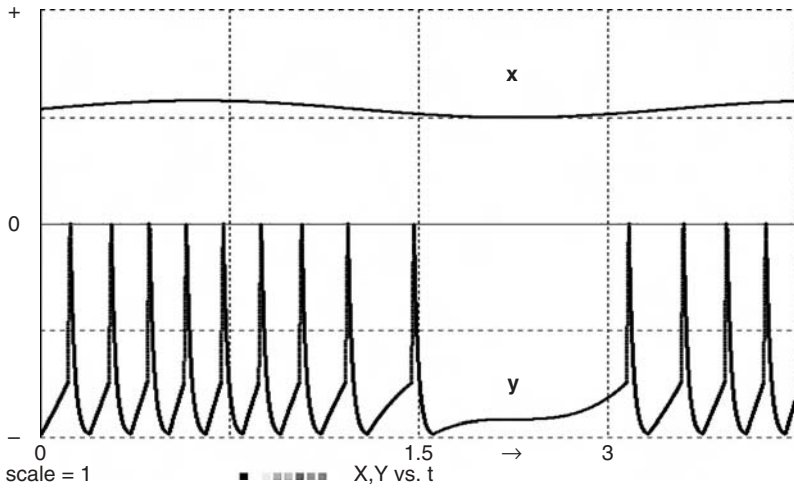
The effective integrator input $ydot$ is produced by a combination of switching functions preceded by a **step** operator that ensures proper numerical integration (Sections 2-9 and 2-11). The crucial point is the use of a difference-equation state variable (Section 2-16) z defined by the recursive assignment (difference equation)



(a)



(b)



(c)

FIGURE 6-14. (a) Pulsed-neuron model, and (b,c) integrate-and-fire time histories.

$$z = \text{swtch}(y - \text{peak} + bb * z) \quad [bb = (1 - y_0) * \text{peak}]$$

We start with the integrator output y at its rest value $y = y_0$ (Figs 6-14a and 6-15). y_0 is small, so that z is negative and the relay-comparator function produces $\dot{y} = c * \dot{y}_{dot1}$. Then for a positive neuron input x :

1. The neuron *integrates its input* x with $\dot{y} = c * \dot{y}_{dot1} = c * x$ until y reaches the firing threshold **fire1**.
2. y then *rises rapidly* with $\dot{y} = c * \dot{y}_{dot1} = c * b_1$, until $y = \text{fire2} > \text{fire1}$.
3. y next *rises less rapidly* with $\dot{y} = c * \dot{y}_{dot1} = c * b_2$, until y reaches the peak value $y = \text{peak} > \text{fire2}$. At this point, z becomes positive.
4. Now, the relay comparator produces $\dot{y} = c * \dot{y}_{dot2}$, and y *decays* with $\dot{y} = c * \dot{y}_{dot2} = -r * c * y$, until y returns to its rest value y_0 .

z then becomes negative again, and the process repeats if $x > 0$.

Step 3 sets the pulse width and can be omitted for about 10% extra speed if thin, pointed pulses (Fig. 6-14c) are satisfactory.

As noted in Section 6-24, the integrate-and-fire model will be followed by linear or nonlinear delay sections representing the pulse propagation along the neuron membrane, for example,

$$d/dt(q) = y - q/TT$$

6-26. Neuron-model Replication

To simulate layers or groups of multiple pulsed neurons, we replicate the model of Figure 6-15 by declaring

STATE $y[n]$ | **ARRAY** $x[n], \dot{y}[n], \dot{y}_{dot1}[n], \dot{y}_{dot2}[n], z[n]$

in the experiment protocol and programming the DYNAMIC-segment lines

Vectr $d/dt \ y = \dot{y}_{dot}$

step

Vector $z = \text{swtch}(y - \text{peak} + bb * z)$ | -- state switch

Vector $\dot{y}_{dot1} = x + (b_1 - x) * \text{swtch}(y - \text{fire1}) + (b_2 - b_1) * \text{swtch}(y - \text{fire2})$

Vector $\dot{y}_{dot2} = -r * y * \text{swtch}(y - y_0)$ | -- decay, refractory period

Vector $\dot{y} = c * \text{comp}(z, \dot{y}_{dot1}, \dot{y}_{dot2})$ (6-50)

```

--
                                PULSED NEURON
-----
display N16 | display C17 | display R
NN = 40000 | DT = 0.000025 | TMAX = 3
-----
c = 2000 | -- integrator gain
a = 0.001 | -- input amplitude
b1 = 0.04 | -- determines pulse rise time
b2 = 0.001 | -- determines pulse width
fire1 = 0.25 | fire2 = 0.95
y0 = 0.01 | -- rest level
peak = 1 | -- pulse amplitude
bb = (1 - y0) * peak | -- precompute for speed
r = 0.03 | -- 1/refractory time constant
ydot = 0
drun
-----
DYNAMIC
-----
x = a * sin(2.5 * t) + a | -- positive input
d/dt y = ydot
step
--
z = swtch(y - peak + bb * z) | -- state switch
--
ydot1 = x + (b1 - x) * swtch(y - fire1) + (b2 - b1) * swtch(y - fire2)
ydot2 = - r * y * swtch(y - y0) | -- decay, refractory time
ydot = c * comp(z, ydot1, ydot2)
-----
X = 40 * x + 0.5 * scale | -- offset for stripchart display
Y = y - scale
dispt X,Y

```

FIGURE 6-15. Complete program for an integrate-and-fire model.

where $\mathbf{bb} = (1 - y_0) * \text{peak}$ is precomputed for speed. To introduce multiple neuron inputs $\mathbf{x}[1]$, $\mathbf{x}[2]$, ... $\mathbf{x}[\mathbf{nx}]$, replace \mathbf{x} with a matrix-vector product $\mathbf{A} * \mathbf{x}$ (Section 3-3).

Pulsed neurons can be programmed to affect not only the activations but also the parameters (threshold levels, integrator gains) of other neurons [31]. To provide different neurons with different parameters, one needs to only declare the desired parameters in the vector expressions of Eq. (6-50) as vectors, as in Sections 3-3 and 3-4. In this way, pulsed neurons may produce much more sophisticated models than static neurons [1,29]. Using

index-shifted vectors in the manner of Sections 6-22 and 6-23 is another intriguing possibility.

The switched differential equations modeling pulsed neurons clearly require more computation than the much simpler neural networks in Sections 6-1 to 6-23. On an inexpensive 2.4-GHz personal computer running Linux, the 12-pulse vectorized integrate-and-fire program, without any delay sections, required 47 s for $n = 1000$ neurons, and 518 s for $n = 10,000$ neurons (which probably strained the small processor's cache memory). The assembly-language version of DESIRE required 23 and 294 s on the same computer.

Our simple integrate-and fire model is only one example; Reference [31] is an excellent review of this subject. The difference-equation scheme can be used with many different models.

REFERENCES

1. W. Gerstner and W. M. Kistler, *Spiking Neuron Models*, Cambridge University Press, 2002.
2. R. Fletcher, *Practical Methods of Optimization*, Wiley, New York, 1987.
3. R. Horst et al., *Introduction to Global Optimization*, Kluwer, 1995.
4. L. Luenberger, *Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA, 1986.
5. M. Galassi et al., *Reference Manual for the GNU Scientific Library (gsl)* (current edition) (<ftp://ftp.gnu.org/gnu/gsl/>) (printed copies can be purchased from Network Theory Ltd. at <http://www.network-theory.co.uk/gsl/manual/>).
6. C. M. Bishop, *Neural Networks for Pattern Recognition*, Oxford University Press, 1995.
7. J. Principe et al., *Neural and Adaptive Systems*, Wiley, New York, 2001.
8. M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, MIT Press, Cambridge, MA, 1995.
9. B. Kosko, *Neural Networks and Fuzzy Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
10. K. Madsen et al., *Methods for Nonlinear Least-squares Problems*, 2nd Ed., Informatics and Mathematical Modeling Group, Tech. University of Denmark, 2004.
11. V. N. Vapnik, *The Nature of Statistical Learning Theory*, Springer, New York, 1995.
12. L. Fausett, *Fundamentals of Neural Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1994.

13. H. Haken, *Synergetic Computers and Cognition*, Springer, New York, 1991.
14. S. Haykin, *Neural Networks*, 2nd Ed., Macmillan, New York, 1998.
15. T. Kohonen, *Self-organization and Associative Memory*, 3rd Ed., Springer, New York, 1989.
16. P. Hecht-Nielsen, The Casasent network, *Proc. IJ CNN*, III-905, 1992.
17. G. A. Korn, *Neural Networks and Fuzzy-logic Control on Personal Computers and Workstations*, MIT Press, Cambridge, MA, 1995.
18. W. T. Miller et al., eds. *Neural Networks in Control*, MIT Press, Cambridge, MA, 1990.
19. S. C. Ahalt et al., Competitive learning algorithms for vector quantization, *Neural Networks*, **3**, 1990, pp. 277–290.
20. S. Grossberg, *The Adaptive Brain* (2 vols.), North-Holland, Amsterdam and New York, 1987.
21. P. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, Reading, MA, 1989.
22. S. Grossberg, *Neural Networks and Natural Intelligence*, MIT Press, Cambridge, MA, 1988.
23. G. A. Carpenter and S. Grossberg, ART 3: Hierarchical search using chemical transmitters, *Neural Networks*, Vol. **3**, 1990, pp. 129–152.
24. G. A. Carpenter, Neural-network models for pattern recognition and associative memory, *Neural Networks*, **2**, 1990, pp. 243–257 (a useful review of earlier work).
25. G. A. Carpenter et al., ART-2A, *Neural Networks*, **4**, 1991, pp. 493–504.
26. (a) G. A. Carpenter, Fuzzy ART, *Neural Networks*, **4**, 1991, pp. 759–771.
(b) G. A. Carpenter et al., Fuzzy ARTMAP, *IEEE Trans. Neural Networks*, **3**, 1992, pp. 698–713.
27. L. Xu et al., Rival Penalized Competitive Learning, *IEEE Transactions on Neural Networks*, **4**, 1993, pp. 636–648.
28. G. A. Korn, New, faster algorithms for supervised competitive learning: counterpropagation and adaptive-resonance functionality, *Neural Processing Letters*, **9**, 1999, pp. 107–117.
29. W. Mass and C. M. Bishop, eds., *Pulsed Neural Networks*, MIT Press, Cambridge, MA, 2001.
30. A. V. Oppenheim, *Discrete-Time Signal Processing*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1989.
31. W. Gerstner, Integrate-and-fire Neurons and Networks, in *The Handbook of Brain Theory and Neural Networks* (M. A. Arbib, ed.), 2nd Ed., MIT Press, Cambridge, MA, 2002.
32. V. N. Vapnik, *Statistical Learning Theory*, Wiley, New York, 1998.

33. Ben Israel and T. N. E. Greville, *Generalized Inverses*, Wiley, New York, 1974.
34. B. Noble and J. W. Daniel, *Applied Linear Algebra*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1977.
35. R. J. Williams and D. Zipser, A learning algorithm for continually running fully recurrent neural networks, *Neural Computation*, **1**, 1989, pp. 270–280.
36. J. L. Elman, Finding structure in time, *Cognitive Science*, **14**, 1990, pp. 179–211.

7

More Applications of Vector Models

A VECTORIZED SIMULATION WITH LOGARITHMIC PLOTS

7-1. The EUROSIM No. 1 Benchmark Problem

One of the EUROSIM benchmark problems posed by Breitenecker and Husinsky [1] models the concentrations **r**, **m**, and **f** of three alkali hydrides under electron bombardment with the state-equation system

$$\begin{aligned} \mathbf{A} &= \mathbf{k} \mathbf{r} * \mathbf{m} * \mathbf{f} - \mathbf{d} \mathbf{r} * \mathbf{r} & \mathbf{B} &= \mathbf{k} \mathbf{f} * \mathbf{f} * \mathbf{f} - \mathbf{d} \mathbf{m} * \mathbf{m} \\ (\mathbf{d} \mathbf{r} / \mathbf{d} \tau) &= \mathbf{A} & (\mathbf{d} \mathbf{m} / \mathbf{d} \tau) &= \mathbf{B} - \mathbf{A} & (\mathbf{d} \mathbf{f} / \mathbf{d} \tau) &= \mathbf{p} - \mathbf{L} \mathbf{f} * \mathbf{f} - \mathbf{A} - 2 * \mathbf{B} \end{aligned}$$

where τ is physical time, not computer time. These are nonlinear differential equations similar to those used in population dynamics (Section 1-12), and also in chemical reaction-rate problems. This benchmark problem is “stiff” or difficult to integrate in the sense that the absolute ratio of the largest to the smallest Jacobian eigenvalue exceeds 120,000 for $\tau = 0$ [2].

7-2. Vectorized Simulation with Logarithmic Plots

Since the solutions for the given coefficient values vary over a wide range, the benchmark problem specified logarithmic scales for the solution **f** and

also for the time. Most of the 25 simulation programs submitted for the benchmark competition [1] solved the differential equations seven times and then obtained logarithmic scales with a plotting program. The vectorized DESIRE program in Figure 7-1 produces all seven solutions in a single simulation run and uses a different approach to obtain logarithmic time scaling.

We relate the computer time variable t to the problem time τ so that

$$\tau = 10t - t_0 \quad (d\tau/dt) = \ln(10)10t - t_0 = tt$$

Multiplication of each given differential equation by tt then produces the new differential-equation model

$$tt = \ln 10 * (10^{(t - t_0)}) \\ d/dt r = A * tt \quad | \quad d/dt m = (B - A) * tt \quad | \quad d/dt f = (p - Lf * f - A - 2 * B) * tt$$

The extra time-scaling operations must execute at each derivative call, not only at the output points. But they serendipitously reduce the *stiffness factor* (ratio of the Jacobian eigenvalues), so that our variable-step/variable-order integration routine automatically uses larger integration steps **DT**. In any case, the exponential time factor tt is common to all seven replicated models and is thus computed only once per derivative call.

For logarithmic scaling of the state variable f , the program directly plots **lgfplus1 = log e * ln(f) + 1**. This assignment needs to be computed only at the sampling points and can thus follow an **OUT** statement (Section 1-6).

Vectorization is not really needed for such a small model. But compared to the all-scalar model in Reference [2], vectorization reduced the benchmark time by a factor of 4 with the display turned off, and by a factor of 2 with the display on.

MODELING FUZZY-LOGIC FUNCTION GENERATORS

7-3. Rule Tables Specify Heuristic Functions

Regression, prediction, and controller-design problems all require construction of a function $y = y(x_1, x_2, \dots)$ that minimizes an error measure or cost. For regression, $y = y(x_1, x_2, \dots)$ is a regression function designed to minimize, say, a mean-square regression error (Section 6-6). In control engineering, $y = y(x_1, x_2, \dots)$ is a controller output that depends on inputs such as servo error and output rate. $y = y(x_1, x_2, \dots)$ must minimize a dynamic-system performance measure such as servo integral square error (Section 1-14).

Regression problems usually yield to numerical methods, but accurate optimization of a nonlinear control system may be difficult. In either case,

fuzzy-set techniques try to design $y = y(x_1, x_2, \dots)$ heuristically by invoking the designer's intuition or accumulated knowledge.

First, consider a function $y = y(x)$ of a single input, and divide the range of x into just a few (typically between 2 and 7) mutually exclusive class intervals, which may have different sizes. The class intervals can be numbered, or they can be given names such as **negative**, **positive**, **very negative**, **near zero**, or **cold**, **warm**, **hot**, and so on. We assign a corresponding small number of numerical function values $y(x)$ by specifying a rule table such as

if x is **negative**, then $y = -1014$
 if x is **near zero**, then $y = 0.2$

Our choice of class intervals and function values, presumably based on intuition or experience, defines a function $y = y(x)$. We can actually try this function on our regression or control problem. But $y(x)$ is a coarsely defined and necessarily discontinuous step function.

One can similarly construct a function $y = y(x_1, x_2)$ of 2 inputs x_1, x_2 . We again divide the ranges of x_1 and x_2 into class intervals (x_1 and x_2 can have different class-interval numbers and/or sizes) and try to invent a two-dimensional rule table such as

if x_1 is **negative** AND x_2 is **very negative** then $y = 1200$
 if x_1 is **negative** AND x_2 is **near zero** then $y = 0$

We have now defined a step function $y = y(x_1, x_2)$ with two inputs. We can add more inputs.

7-4. Fuzzy-set Logic

Fuzzy-set techniques also invoke heuristic rule tables but produce at least piecewise-continuous functions instead of coarse step functions.

(a) Fuzzy Sets and Membership Functions

We replace our input class intervals with similarly labeled abstract fuzzy sets of x values, for example, **very negative**, **negative**, **near zero**, **positive**, and **very positive**. Membership of a given input value $x = X$ in a fuzzy set E is defined by a nonnegative membership function $M(E | x)$ that measures the degree to which the value X "belongs" to the fuzzy set. We regard the proposition that a measured value X of x belongs to a fuzzy set E as an abstract event with the "fuzzy truth value" $M(E | X)$.

Figures 7-2 and 7-3 show examples. Note that membership functions can overlap, which means that a value \mathbf{x} of \mathbf{x} can “belong” to more than one fuzzy set. Classical truth values associated with mutually exclusive (“crisp”) class intervals can be regarded as special membership functions equal to 1 on a single class interval and 0 elsewhere. Singleton fuzzy sets have membership functions that equal 1 for a single “support value” $\mathbf{x} = \mathbf{X}$ and are 0 elsewhere.

(b) Fuzzy Intersections and Unions

We next define membership functions for (1) the abstract event that \mathbf{X} belongs to the fuzzy set **E1 AND** to the fuzzy set **E2**, and (2) for the abstract event that \mathbf{X} belongs to the fuzzy set **E1 OR** to the fuzzy set **E2**:

$$M(\mathbf{E1 AND E2} \mid \mathbf{x}) \equiv M(\mathbf{E1} \mid \mathbf{x}) m(\mathbf{E2} \mid \mathbf{x}) \quad (\text{fuzzy intersection})$$

$$M(\mathbf{E1 OR E2} \mid \mathbf{x}) \equiv M(\mathbf{E1} \mid \mathbf{x}) + m(\mathbf{E2} \mid \mathbf{x}) \quad (\text{fuzzy union})$$

We call this *product/sum logic*.¹

(c) Joint Membership Functions

For multiple input variables $\mathbf{x1}, \mathbf{x2}, \dots$ we define *multidimensional* fuzzy sets **E** by membership functions $M(\mathbf{E} \mid \mathbf{x1}, \mathbf{x2}, \dots)$. We extend product/sum logic to relate multidimensional fuzzy sets to intersections of lower-dimensional fuzzy sets in terms of joint membership functions such as

$$M(\mathbf{E} \mid \mathbf{x1}, \mathbf{x2}) \equiv M(\mathbf{E1 AND E2} \mid \mathbf{x1}, \mathbf{x2}) \equiv M(\mathbf{E1} \mid \mathbf{x1}) M(\mathbf{E2} \mid \mathbf{x2})$$

One can also define unions of fuzzy sets in the $\mathbf{x1}$ and $\mathbf{x2}$ domains, as in

$$M(\mathbf{E1 OR E2} \mid \mathbf{x1}, \mathbf{x2}) \equiv M(\mathbf{E1} \mid \mathbf{x1}) + M(\mathbf{E2} \mid \mathbf{x2})$$

(d) Normalized Fuzzy-set Partitions

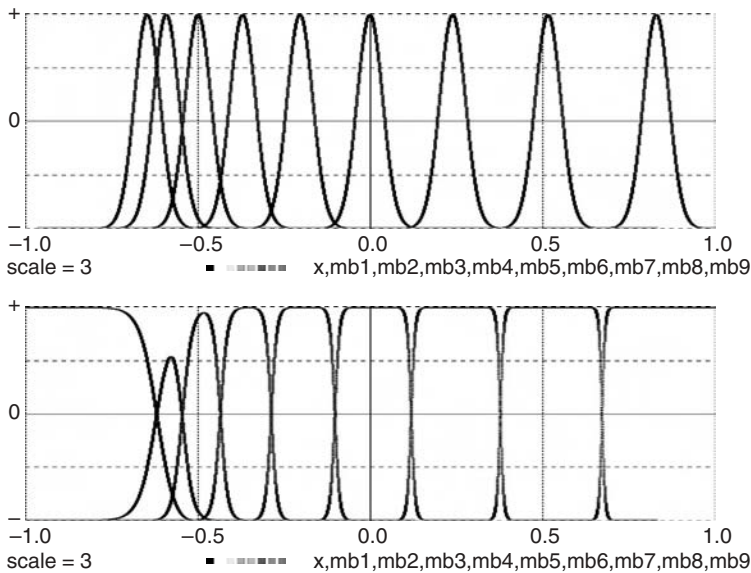
N fuzzy sets **E1, E2, ..., EN** form a fuzzy-set partition “covering” the domain of $\mathbf{x} \equiv (\mathbf{x1}, \mathbf{x2}, \dots, \mathbf{xn})$ if at least one of the fuzzy-set membership functions $M(\mathbf{Ei} \mid \mathbf{x})$ differs from 0 for every \mathbf{x} . In the following sections, we shall always use product/sum logic and normalized fuzzy-set partitions whose membership functions add up to 1 for every value of \mathbf{x} (see also

¹ Union and intersections can be alternatively defined by *min/max logic*, as in

$$M(\mathbf{E1 AND E2} \mid \mathbf{x1}, \mathbf{x2}) \equiv \min[M(\mathbf{E1} \mid \mathbf{x1}), M(\mathbf{E2} \mid \mathbf{x2})]$$

$$M(\mathbf{E1 OR E2} \mid \mathbf{x1}, \mathbf{x2}) \equiv \max[M(\mathbf{E1} \mid \mathbf{x1}), M(\mathbf{E2} \mid \mathbf{x2})]$$

Min/max logic simplifies inexpensive fixed-point controllers but usually slows floating-point computations.



-- GAUSSIAN FUZZY-SET MEMBERSHIP FUNCTIONS

-- submodel for computing membership functions

--

ARRAY X\$[1], mb\$[1] | -- dummy-argument arrays

SUBMODEL fuzzmemb(N\$, X\$, mb\$, input\$)

Vector mb\$ = exp(-b * (X - 2 * x)^2)

DOT sum = mb\$ * 1 | ss = 1/sum | -- normalize

Vector mb\$ = ss * mb\$

end

scale = 3 | -- declare arrays

N = 9 | b = 10 | -- number and spread

ARRAY X[N] | -- peak ordinates

ARRAY mb[N] | -- limiter functions

-- define fuzzy-set spacing

x0 = - scale

for i = 1 to N | X[i] = (i^2 - N)/N + x0 | next

NN = 10000 | TMAX = 1 | DT = 0.0001

--

x = - scale | drun

DYNAMIC

d/dt x = 2 * scale | -- display sweep

invoke fuzzmemb[N,X,mb,x]

Vector mb = 6 * mb - scale | -- scaled, offset display

FIGURE 7-2. Fuzzy-set membership functions before and after normalization, and a program for experiments with different numbers and spacings of fuzzy sets. Note the effect of normalization at the ends of the range. With increasing spacing, the un-normalized fuzzy sets approximate singleton fuzzy sets, and the normalized fuzzy sets approximate conventional class intervals.

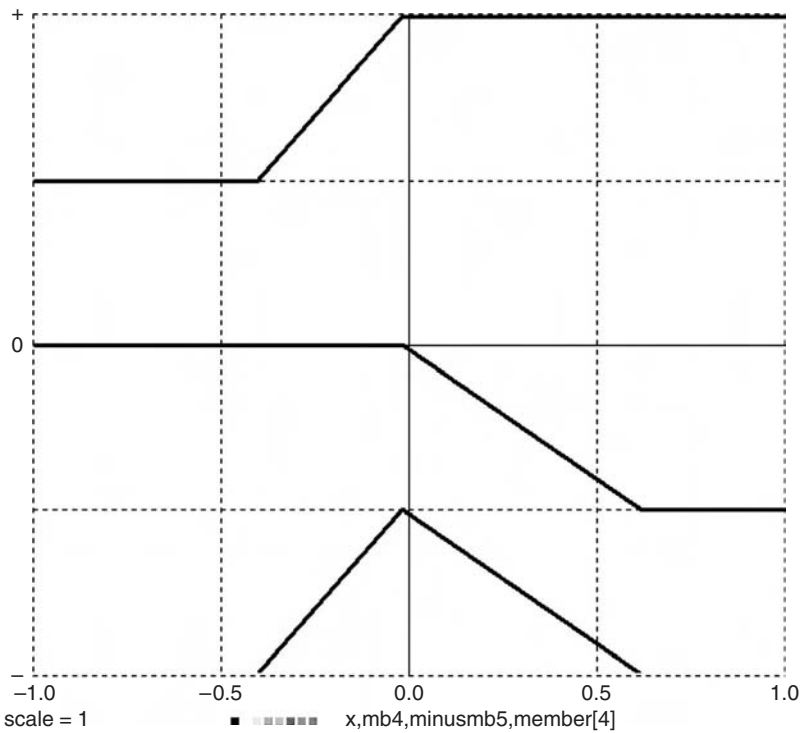


FIGURE 7-3a. Generating a triangle function as a difference of two limiter functions (S. Geva and J. Sitte, *IEEE Trans. Neural Networks*, **3**, 1994, pp. 622–624).

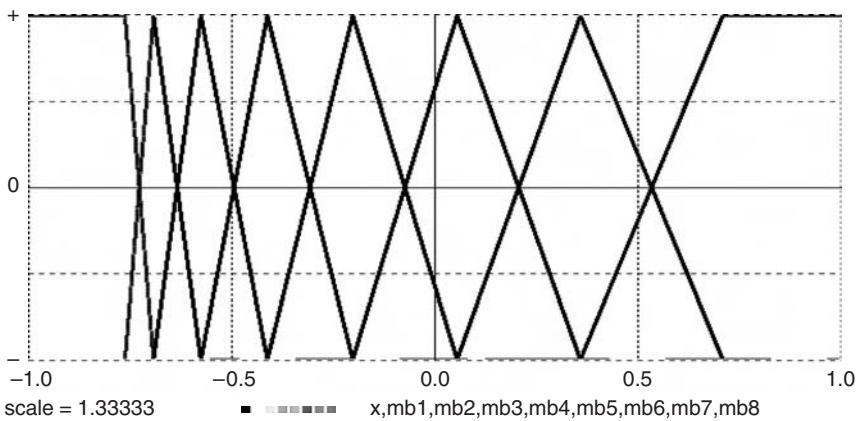


FIGURE 7-3b. N overlapping triangle functions form a very useful normalized fuzzy-set partition.

Section 7-7). This implies that all our membership functions range between 0 and 1.²

Membership functions of normalized partitions for individual variables $\mathbf{x}_1, \mathbf{x}_2, \dots$ and for combinations of such variables can be simply multiplied to define normalized partitions of higher-dimensional domains.³

7-5. Fuzzy-set Rule Tables and Function Generators

Rule tables can relate output fuzzy-set memberships rather than numerical values to input fuzzy-set memberships, for example, if \mathbf{x} is **very positive** then \mathbf{y} is **hot**. One can then define fuzzy-set membership functions $\mathbf{M}_1(\mathbf{E}_{i1} \mid \mathbf{x}_1)$, $\mathbf{M}_2(\mathbf{E}_{i2} \mid \mathbf{x}_2)$, ... for each function-generator input $\mathbf{x}_1, \mathbf{x}_2, \dots$, membership functions $\mathbf{M}(\mathbf{E}_i \mid \mathbf{y})$ for the function-generator output \mathbf{y} , and joint input–output membership functions

$$\mathbf{M}(\mathbf{E}_{i1}, \mathbf{E}_{i2}, \dots; \mathbf{E}_i \mid \mathbf{x}_1, \mathbf{x}_2, \dots; \mathbf{y}) = \mathbf{M}_1(\mathbf{E}_{i1} \mid \mathbf{x}_1) \mathbf{M}_2(\mathbf{E}_{i2} \mid \mathbf{x}_2) \dots \mathbf{M}(\mathbf{E}_i \mid \mathbf{y})$$

with $i_1 = 1, 2, \dots, \mathbf{N}_1$; $i_2 = 1, 2, \dots, \mathbf{N}_2, \dots$; $i = 1, 2, \dots, \mathbf{N}$. The fuzzy-set partition sizes $\mathbf{N}_1, \mathbf{N}_2, \dots, \mathbf{N}$ are usually small (between 2 and 7).

We now define a different type of rule table, namely, an $(\mathbf{N}_1 \mathbf{N}_2 \dots \mathbf{N})$ -dimensional vector whose components, similar to those of $\mathbf{M}(\mathbf{E}_{i1}, \mathbf{E}_{i2}, \dots; \mathbf{E}_i \mid \mathbf{x}_1, \mathbf{x}_2, \dots; \mathbf{y})$ can be ordered by their index combinations i_1, i_2, \dots, i . We heuristically set the rule-table entries to 1 or 0 when we consider the corresponding input–output combination as possible or impossible. These rules normally associate a single fuzzy set in the output domain with each combination of input sets, but each output set can be selected by more than one input combination.

The resulting fuzzy output set is the fuzzy union of all joint fuzzy input–output sets that are not eliminated by the rule table. With product/sum logic, its membership function $\mathbf{P}(\mathbf{y})$ is the sum of the corresponding joint membership functions $\mathbf{M}(\mathbf{E}_{i1}, \mathbf{E}_{i2}, \dots; \mathbf{E}_i \mid \mathbf{x}_1, \mathbf{x}_2, \dots; \mathbf{y})$. This is all fuzzy logic can tell us about \mathbf{y} . Obtaining a usable “crisp” function-generator output $\mathbf{y}(\mathbf{x}_1, \mathbf{x}_2, \dots)$ requires a *heuristic defuzzification assumption* such as

$$\mathbf{y} = \frac{\int \mathbf{y} \mathbf{P}(\mathbf{y}) d\mathbf{y}}{\int \mathbf{P}(\mathbf{y}) d\mathbf{y}} \quad (\text{centroid defuzzification})$$

² In this context, we can define the logical complement \mathbf{E}' of \mathbf{E} by its membership function $\mathbf{M}(\mathbf{E}' \mid \mathbf{x}) = 1 - \mathbf{M}(\mathbf{E} \mid \mathbf{x})$.

³ Note that this requires product/sum logic.

or alternatively

$$\mathbf{y} = \mathbf{ymax} \text{ where } \mathbf{P}(\mathbf{ymax}) \text{ is a maximum of } \mathbf{P}(\mathbf{y})$$

(maximum defuzzification)

Here, integrals and maximum are taken over the range of \mathbf{y} . Such techniques are further discussed in Reference [3].

7-6. Simplified Function Generation with Fuzzy Basis Functions

The general fuzzy-set technique outlined in Section 7-5 is complicated and involves rather arbitrary defuzzification assumptions. Although DESIRE allows programming this general method [4], many practical regression and controller-design problems yield to a much simpler procedure.

Assume that we have a normalized fuzzy-set partition of the input-variable domain and a rule table that assigns a “crisp” function output value $\mathbf{y}[i]$ (not an output fuzzy set) to each fuzzy set \mathbf{E}_i of the input-variable partition. We then employ the weighted sum

$$\mathbf{y}(\mathbf{x}) = \mathbf{y}[1] \mathbf{M}(\mathbf{E}_1 | \mathbf{x}) + \mathbf{y}[2] \mathbf{M}(\mathbf{E}_2 | \mathbf{x}) + \dots + \mathbf{y}[\mathbf{N}] \mathbf{M}(\mathbf{E}_\mathbf{N} | \mathbf{x}) \quad (7-1)$$

as a regression or controller function designed to “fit” our rule table. \mathbf{x} can be a multidimensional set of variables $\mathbf{x} \equiv \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ [14].

The \mathbf{N} normalized fuzzy-set membership functions $\mathbf{mb}[i] = \mathbf{M}(\mathbf{E}_i | \mathbf{x})$ are often called fuzzy basis functions. Section A-2 in the Appendix describes their application to neural networks, where they are used much like radial basis functions (Section 7-13). The function-generator output $\mathbf{y}(\mathbf{x})$ is determined by the heuristic rule-table entries $\mathbf{y}[i]$ and by the number and shape of the fuzzy basis functions, which are normally continuous or piecewise continuous functions of \mathbf{x} .⁴

7-7. Vector Models of Fuzzy-set Partitions

(a) Gaussian Bumps—Effects of Normalization

We begin with functions of a one-dimensional argument \mathbf{x} . To generate \mathbf{N} bump-shaped fuzzy basis functions $\mathbf{mb}[1], \mathbf{mb}[2], \dots, \mathbf{mb}[\mathbf{N}]$ centered on \mathbf{N} given values $\mathbf{X} = \mathbf{X}[1], \mathbf{X}[2], \dots$ of \mathbf{x} , a DESIRE program can declare vectors \mathbf{X} and \mathbf{mb} with

ARRAY $\mathbf{X}[\mathbf{N}], \mathbf{mb}[\mathbf{N}]$

⁴ With random input \mathbf{x} , one can consider the \mathbf{N} fuzzy-set memberships as abstract random events with conditional probabilities $\mathbf{M}(\mathbf{E}_i | \mathbf{x})$. The expression (7-1) is then the expected value of $\mathbf{y}[i]$. This, in fact, models a well-known hardware technique, namely, function-generator interpolation using dither-noise injection and averaging.

and assign **X[i]** values in the experiment-protocol script. The DYNAMIC-segment lines

```
Vector mb = a * exp(- b*(X - x)^2)  
DOT sum = mb * 1 | ss = 1/sum | Vector mb = ss * mb
```

then produce **N** Gaussian bumps and divide them by their sum to normalize them (Fig. 7-2). The effect of normalization on the first and last fuzzy-set membership function is realistic and intuitively plausible. The normalized Gaussian bumps in Figure 7-2b have different amplitudes since they have the same “spread” parameter **b** but are not uniformly spaced. In the following sections, we exhibit membership functions whose spread changes with their spacing [13, 14].

(b) Triangle Functions

Suitably overlapping triangle-shaped functions **mb[i]** with unity peaks at **x = x[1], x[2], ..., x[N]** (Fig. 7-3b) produce particularly useful normalized fuzzy-set partitions. They can implement exact linear interpolation between adjacent rule-table function values.

To generate the desired triangle functions of the input **x**, we first use index shifting (Section 3-6) to create **N** limiter functions (Section 2-8a)

```
Vector mb = SAT((X - x)/(X - X{1}))
```

Vector components with index values shifted below 1 and above **N** are automatically replaced by zero, and we save the two end values

```
Mbb = mb[1] | mcc = mb[N - 1]
```

for use later in the program. Pairwise subtraction of index-shifted limiter functions with

```
Vector mb = mb{-1} - mb
```

(Fig. 7-3a) then produces the desired **N** overlapping triangle functions **mb[i]** if we overwrite **mb[1]** and **mb[N]** with

```
mb[1] = 1 - mbb | mb[N] = mcc
```

This algorithm is twice as fast as the original algorithm in References [4] and [5] and also uses only half the memory required by the earlier program.

The procedure can be stored as a library submodel (Section 3-17):

```

ARRAY X$[1], mb$[1] | -- dummy-argument arrays
--
SUBMODEL fuzzmemb(N$, X$, mb$, input$)
  Vector mb$ = SAT((X$ - input$)/(X$ - X${1}))
  mbb = mb$[1] | mcc = mb$[N$ - 1]
  Vector mb$ = mb${-1} - mbb
  mb$[1] = 1 - mbb | mb$[N$] = mcc
end

```

for function generation, regression, and control-system simulation (Section 7-9).

(c) Smooth Fuzzy Basis Functions

It is easy to replace the piecewise-continuous triangles in Section 7-7b with differentiable functions. The soft-limiting DESIRE library function

$$\text{sigmoid}(q) \equiv 1/(1 + \exp(-q))$$

should be used instead of the hard-limiting **SAT**(q) function in Section 7-7b (Fig. 7-3a). In regression applications, such nonlinear curve fitting may allow using fewer basis functions. The membership-function shape had almost no effect on the controller in Section 7-9.

7-8. Vector Models for Multidimensional Fuzzy-set Partitions

Given normalized fuzzy-set partitions for each of two independent input variables **x1**, **x2**, with arrays (vectors) of membership functions

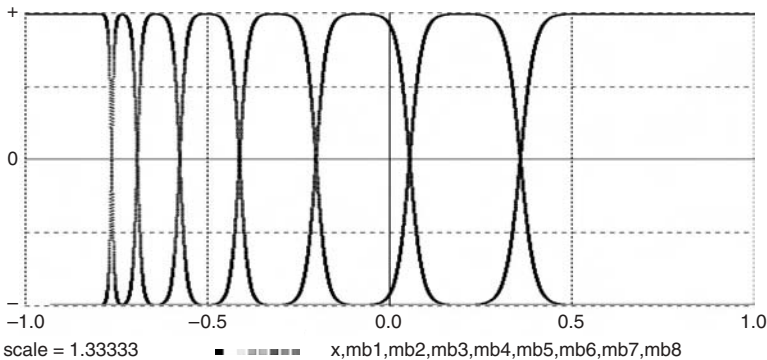


FIGURE 7-4. A normalized fuzzy-set partition obtained with differences of soft limiter functions.

$$\begin{aligned}\mathbf{mb1} &\equiv [\mathbf{M}(\mathbf{E11} \mid \mathbf{x1}), \mathbf{M}(\mathbf{E12} \mid \mathbf{x1}), \dots, \mathbf{M}(\mathbf{E1N1} \mid \mathbf{x1})] \\ \mathbf{mb2} &\equiv [\mathbf{M}(\mathbf{E21} \mid \mathbf{x2}), \mathbf{M}(\mathbf{E22} \mid \mathbf{x2}), \dots, \mathbf{M}(\mathbf{E2N1} \mid \mathbf{x2})]\end{aligned}$$

the **N1N2** joint membership functions (Section 7-4c)

$$\mathbf{MB}[i, k] \equiv \mathbf{M}(\mathbf{E1i} \mid \mathbf{x1}) \mathbf{M}(\mathbf{E1k} \mid \mathbf{x1}) \quad (i = 1, 2, \dots, \mathbf{N1}; k = 1, 2, \dots, \mathbf{N2})$$

form a normalized fuzzy-set partition covering the domain of joint observations **x1**, **x2**. The **N1 × N2** matrix **MB** is neatly produced by the DYNAMIC-segment matrix assignment (Section 3-10)⁵

$$\mathbf{MATRIX MB} = \mathbf{mb1} * \mathbf{mb2}$$

Our experiment-protocol scrip can define an **N1N2**-dimensional membership-function vector **mb** equivalent to the **N1 × N2** matrix **MB** by declaring (Section 3-11)

$$\mathbf{ARRAY mb} = \mathbf{MB}[\mathbf{N1}, \mathbf{N2}]$$

This will permit us to compute the desired function (7-1) as a simple inner product (Section 7-9).

This procedure is readily extended to three or more dimensions. For three input variables **x1**, **x2**, **x3**, for example, one would declare

$$\mathbf{ARRAY mb} = \mathbf{MB}[\mathbf{N1}, \mathbf{N2}], \mathbf{mmb} = \mathbf{MMB}[\mathbf{N1} * \mathbf{N2}, \mathbf{N3}]$$

in the experiment-protocol script and then assign

$$\mathbf{MATRIX MB} = \mathbf{mb1} * \mathbf{mb2} \mid \mathbf{MATRIX MMB} = \mathbf{mb} * \mathbf{mb3}$$

in a DYNAMIC program segment.

7-9. Example: Fuzzy-logic Control of a Servomechanism

(a) Problem Statement

Recalling the servomechanism model in Section 1-14, we replace its linear controller function

$$\mathbf{voltage} = -k * \mathbf{error} - r * \mathbf{xdot}$$

⁵ If min/max fuzzy-set logic is preferred, the DESIRE matrix assignment **MATRIX MB = mb1 & mb2** produces matrix elements **min[M(E1i | x1), M(E1k | x1)]**. But these joint membership functions would have to be renormalized.

by a nonlinear fuzzy-logic controller function **voltage(e, xdot)** of the servo error **e** and the output rate **xdot**. We define **N1 = 5** fuzzy sets (**very negative**, **negative**, **small**, **positive**, and **very positive**) for **e** and **N2 = 5** fuzzy sets for **xdot** with triangle membership functions such as those in Section 7-7b. We will use the **N1N2 = 25** products of these triangle functions as joint fuzzy-set membership functions for **e** and **xdot**, assign heuristic rule-table values **voltage[k]** to each fuzzy set, and invoke Eq.(7-1) to produce the controller output **voltage(e, xdot)**.

(b) Experiment Protocol and Rule Table

The experiment-protocol script in Figure 7-5a first defines the triangle-function submodel described in Section 7-7b. We then declare triangle-peak-abscissa vectors **xx1**, **xx2** and membership-function vectors **mb1**, **mb2** for the servo error **e** and the output rate **xdot** with

```

N1 = 5
ARRAY xx1[N1]      |  --  peak locations for e
ARRAY mb1[N1]      |  --  membership functions for e
--
N2 = 5
ARRAY xx2[N2]      |  --  peak locations for xdot
ARRAY mb2[N2]      |  --  membership functions for xdot

```

We next declare the **N1 × N2** joint-membership matrix **M12** and an equivalent **N1N2**-dimensional joint-membership vector **m12**, as in Section 7-8:

```

ARRAY M12[N1, N2] = m12      |  --  joint memberships

```

The **N1 × N2** rule-table vector **ruletabl** is declared with

```

ARRAY ruletabl[N1 * N2] | --  controller rule table

```

We use data/read assignments to fill the triangle-peak-location arrays **xx1**, **xx2** with the values

```

-2emax, 0.05emax, 0, 0.05emax, 2emax  for e
-2xdotmax, -0.5xdotmax, 0, 0.5 dotmax, 2xdotmax  for xdot

```

where **emax = xdotmax = 1**. We fill the rule-table array **ruletabl** as follows:

if e is very negative	-8k-8r, -8k-r, -8k, -8k+r, -8k+8r
if e is negative	-2k-2r, -2k-r, -5k, -2k+r, -2k+2r
if e is small	-2r, -0.08r, 0, 0.08r, 2r
if e is positive	2k-2r, 2k-r, 5k, 2k+r, 2k+2r
if e is very positive	8k-8r, 8k-r, 8k, 8k+r, 8k+8r

Successive entries in each row refer to **xdot** = **very negative**, **negative**, **small**, **positive**, **very positive**, and **k** = 0.35 and **r** = 2. Note that we wrote each rule-table entry in the form $\alpha k + \beta r$. αk is our intuitive guess at the controller-output contribution due to **e**, and βr is our idea of the contribution due to **xdot**.

Our choices of peak-location abscissas and rule-table entries express a heuristic guess for a controller design. In this example, we decided to use larger-than-linear controller gains for large servo errors and little or no damping for very small servo errors. Our results (Fig. 7-6a) did produce a better noise-following and step-input response than a linear controller.

The remainder of the experiment-protocol script in Figure 7-5a sets system parameters for the fuzzy-logic-controlled servomechanism and also for a similar servo using a linear controller. The script then calls a simulation run to display the time histories of both servomechanisms for comparison (Fig. 7-6a). Another simulation run exercises a second DYNAMIC program segment to display the fuzzy-set membership functions for the servo error **e**.

(c) DYNAMIC Program Segment and Results

The DYNAMIC program segment in Figure 7-5b invokes the triangle-function submodel described in Section 7-7b twice to generate the fuzzy-set membership functions **mb1[k]** and **mb2[k]** for **e** and **xdot**. The desired controller output voltage **voltage(e, xdot)** is then produced as a **DOT** (Section 3-7a):

DOT Voltage = ruletabl * m12

Figure 7-6a shows the servo response to a random-noise input together with that obtained with an optimized linear controller. Results are comparable to those produced with an early version of DESIRE in References [4,5], but our new program is simpler and faster. In practice, these experiments must be repeated with different signal amplitudes, since the control system is non-linear.

FIGURE 7-5a. The experiment-protocol script for the fuzzy-logic-controlled servomechanism defines the triangle-function submodel, sets up triangle-peak abscissas, rule table, and system parameters, and calls a simulation run. Another simulation run uses a second DYNAMIC program called **members** to display the fuzzy-set membership functions.

```

--          FUZZY-LOGIC-CONTROLLED SERVOMECHANISM
--          also simulates a similar linear servo for comparison
-----
--          triangle-function partition
ARRAY X$[1], mb$[1] | -- dummy-argument arrays
SUBMODEL fuzzmemb(N$, X$, mb$, input$)
  Vector mb$ = SAT((X$ - input$)/(X$ - X${1}))
  mbb = mb$[1] | mcc = mb$[N$ - 1]
  Vector mb$ = mb${-1} - mbb
  mb$[1] = 1 - mbb | mb$[N$] = mcc
end
-----
--          declare arrays for e, xdot fuzzy-set membership functions
--
N1 = 5
ARRAY xx1[N1] | -- peak locations for e
ARRAY mb1[N1] | -- membership functions for e
--
N2 = 5
ARRAY xx2[N2] | -- peak locations for xdot
ARRAY mb2[N2] | -- membership functions for xdot
--
ARRAY M12[N1, N2] = m12 | -- joint memberships
ARRAY ruletabl[N1 * N2] | -- controller rule table
-----
--          read membership-peak abscissas
emax = 1 | xdotmax = 1
data -2*emax, -0.05 * emax, 0, 0.05 * emax, 2 * emax
data -2*xdotmax, -0.5*xdotmax, 0, 0.5*xdotmax, 2*xdotmax
read xx1,xx2
-----
A = 1.5 | w = 1
B = 300 | maxtrq = 1 | g1 = 10000 | -- servo parameters
g2 = 2 | R = 0.6
k = 0.3500 | r = 2 | -- fuzzy-controller parameters
kk = 10 | rr = 0.1500 | -- linear-controller parameters
-----
--          rule table
data -8*k-8*r, -8*k-r, -8*k, -8*k+r, -8*k+8*r | -- high gain
data -2*k-2*r, -2*k-r, -5*k, -2*k+r, -2*k+2*r | -- for large errors
data -2*r, -0.08*r, 0, 0.08 * r, 2*r | -- ... and no damping
data 2*k-2*r, 2*k-r, 5*k, 2*k+r, 2*k+2*r | -- for small errors
data 8*k-8*r, 8*k-r, 8*k, 8*k+r, 8*k+8*r
read ruletabl
-----
NN = 4000 | TMAX = 10 | DT = 0.001 | scale = 0.08
p = A * ran() | -- must initialize noise!
drun | -- make a run
write "type go to see membership functions" | STOP
-----
DT = 0.00001 | NN = 40000
scale = 5 | TMAX = 0.5
e = -2.5 | -- start of display sweep
drun members | -- show the membership functions

```

```

-----
d/dt pp = -w * pp + p | d/dt u = -w * u + pp | -- low-pass noise
e = x - u | -- servo error
-- compute membership functions for e and xdot
--
invoke fuzzmemb(N1,xx1,mb1,e) | -- fuzzy sets for e
invoke fuzzmemb(N2,xx2,mb2,xdot) | -- fuzzy sets for xdot
--
MATRIX M12 = mb1 * mb2 | -- make joint membership functions
DOT Voltage = ruletabl * m12 | -- rule-table defuzzification
--
d/dt V = -B * V + g1 * Voltage | -- motor-field buildup
torque = -maxtrq * tanh(g2 * V/maxtrq) | -- servo torque
d/dt x = xdot | d/dt xdot = torque - R * xdot | -- servo dynamics
-----
-- linear servo for comparison
ee = xx - u | -- servo error
VOLTAGE = -kk * ee - rr * xxdot | -- linear controller
d/dt VV = -B * VV + g1 * VOLTAGE | -- motor-field buildup
--
Torque = maxtrq * tanh(g2 * VV/maxtrq) | -- motor torque
d/dt xx = xxdot | d/dt xxdot = Torque - R * xxdot | -- dynamics
--
OUT
p = A*ran() | -- noise is sampled
-----
label members
d/dt e = 2 * scale | -- display sweep
invoke fuzzmemb[N1, xx1, mb1, e) | -- fuzzy sets for e
Vector mb1 = 7.5 * mb1 - scale | -- scale, offset display of mb1

```

FIGURE 7-5b. DYNAMIC program segments for the fuzzy-logic controller. The main DYNAMIC segment generates time histories. An extra DYNAMIC program segment displays the fuzzy-set membership functions for the servo error e .

PARTIAL DIFFERENTIAL EQUATIONS

7-10. The Method of Lines

The numerical method of lines (MOL) reduces a partial differential equation to a set of ordinary differential equations [6–10]. MOL is not the best general-purpose method for solving partial differential equations; finite-difference programs are more general and are usually more convenient and accurate. But MOL is often attractive for process-control simulation, because MOL-generated ordinary differential equations representing reactors or heat exchangers are simply solved together with the ordinary differential equations modeling the rest of the control system.

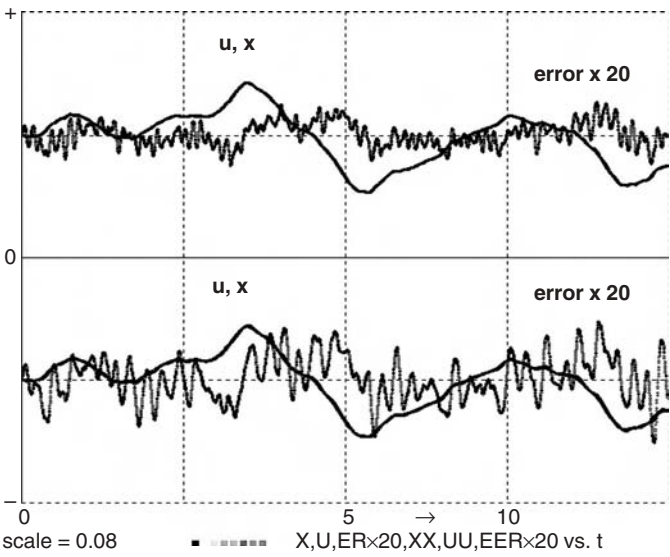


FIGURE 7-6a. Noise-input response of the same servomechanism with a fuzzy controller (top) and a linear controller (bottom).

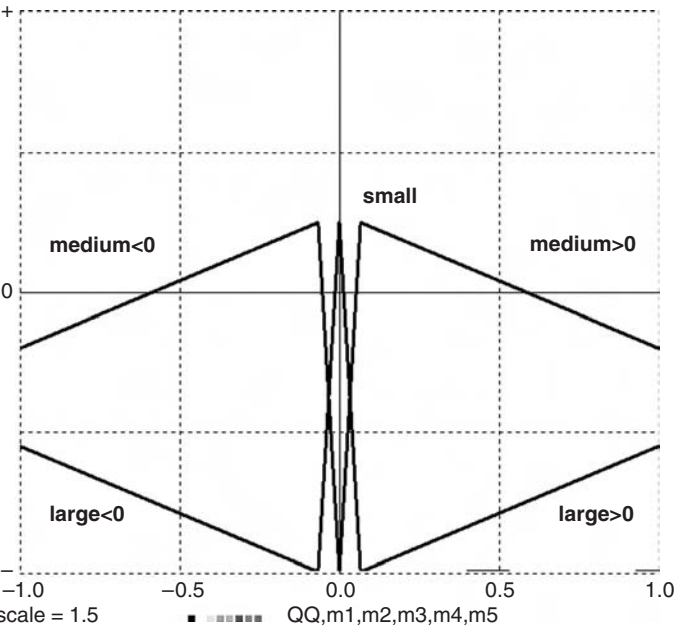


FIGURE 7-6b. The five servo-error fuzzy-set membership functions for small error values. The narrow membership function in the center is used to suppress servo damping for small servo errors.

7-11. The Vectorized Method of Lines

(a) Introduction

The simplest partial-differential-equation problems involve functions $\mathbf{u} = \mathbf{u}(\mathbf{t}, \mathbf{x})$ of the time \mathbf{t} and one space coordinate \mathbf{x} . We will use subscript notation for partial derivatives, as in

$$\partial \mathbf{u} / \partial \mathbf{t} \equiv \mathbf{u}_t \quad \partial \mathbf{u} / \partial \mathbf{x} \equiv \mathbf{u}_x \quad \partial^2 \mathbf{u} / \partial \mathbf{x}^2 \equiv \mathbf{u}_{xx} \quad \dots$$

A useful example is the one-dimensional heat-conduction equation or diffusion equation

$$\mathbf{u}_t = \mathbf{u}_{xx} \quad (7-2)$$

satisfied by the temperature $\mathbf{u} = \mathbf{u}(\mathbf{t}, \mathbf{x})$ in a uniform rod extending from $\mathbf{x} = 0$ to $\mathbf{x} = \mathbf{L}$. We want to find the time histories of $\mathbf{u}(\mathbf{x}, \mathbf{t}) = \mathbf{u}[1], \mathbf{u}[2], \dots, \mathbf{u}[\mathbf{n}]$ at \mathbf{n} uniformly spaced points $\mathbf{x}[1] = 0, \mathbf{x}[2], \dots, \mathbf{x}[\mathbf{n}] = \mathbf{L}$ along the rod.

MOL replaces \mathbf{u}_{xx} with one of several possible difference approximations, say $\{\mathbf{u}[\mathbf{i} - 1] - 2\mathbf{u}[\mathbf{i}] + \mathbf{u}[\mathbf{i} + 1]\} / \mathbf{DX}^2$ and then solves the resulting system

$$(\mathbf{d}/\mathbf{dt})\mathbf{u}[\mathbf{i}] = \{\mathbf{u}[\mathbf{i} - 1] - 2\mathbf{u}[\mathbf{i}] + \mathbf{u}[\mathbf{i} + 1]\} / \mathbf{DX}^2 \quad (\mathbf{i} = 1, 2, \dots, \mathbf{n})$$

of \mathbf{n} ordinary differential equations for $\mathbf{x}[1], \mathbf{x}[2], \dots$. Vectorization represents this system as a single vector differential equation. Reference [9] shows how boundary values of the $\mathbf{u}[\mathbf{i}]$ can be set for given boundary conditions, but this is a problem-specific and error-prone procedure.

(b) Using Differentiation Operators

Schiesser [6] replaced *ad hoc* procedures for selecting difference approximations and setting initial conditions with a systematic approach. He declared separate \mathbf{n} -dimensional arrays $\mathbf{ux}, \mathbf{uxx}, \dots$ for the space derivatives $\mathbf{u}_x, \mathbf{u}_{xx}, \dots$ and defined a Fortran function **DDx** that operates on \mathbf{u} to produce \mathbf{ux} , on \mathbf{ux} to produce \mathbf{uxx} , and so on:

$$\mathbf{ux} = \mathbf{DDx}(\mathbf{u}) \quad \mathbf{uxx} = \mathbf{DDx}(\mathbf{ux}) \quad \dots$$

We will implement such space differentiations with a submodel (Section 3-17) [9]. DESIRE submodels do not impose any runtime function-call overhead and can be stored for reuse. Table 7-1 lists useful submodels for second- and fourth-order central-difference derivative approximations.

The experiment-protocol script in Figure 7-7 declares an \mathbf{n} -dimensional state vector \mathbf{u} and \mathbf{n} -dimensional vectors \mathbf{ux} and \mathbf{uxx} with

```
STATE u[n] | ARRAY ux[n], uxx[n]
```

Table 7-1 Submodels for Schiesser's Partial-Derivative Operators*(a) Second-Order Central-Difference Approximation*

To relate an array $\mathbf{v} \equiv (v[1], v[2], \dots, v[n\$])$ to the corresponding derivative array $\mathbf{vx} \equiv (vx[1], vx[2], \dots, vx[n\$])$, use

$$\begin{aligned} vx[i] &= (v[i+1] - v[i-1])/2DX & (i = 2, 3, \dots, n\$ - 1) \\ vx[1] &= (-3v[1] + 4v[2] - v[3])/2DX & vx[n\$] = (3v[n\$] - 4v[n\$ - 1] + v[n\$ - 2])/2DX \end{aligned}$$

This is implemented with the DESIRE submodel

```
SUBMODEL DDx(n$, bb$, v, vx)
  Vector vx = (v{1} - v{-1}) * bb$
  vx[1] = (-3 * v[1] + 4 * v[2] - v[3]) * bb$      (
  vx[n$] = (3 * v[n$] - 4 * v[n$ - 1] + v[n$ - 2]) * bb$
end                                                    [set bb$ = 1/(2 DX)]
```

Note that the assignments to the end values $vx[1]$ and $vx[n\$]$ overwrite the end values of the vector assignment. The index-shift operation also automatically sets $v[i] = 0$ for $i < 1$ or $i > n\$$ (Section 3-6).

(b) Fourth-order Central-difference Approximation

The corresponding fourth-order submodel is [6,9]

```
SUBMODEL DDx(n$, bb$, v, vx)
  Vector vx = (2 * v{-2} - 16 * v{-1} + 16 * v{1} - 2 * v{2}) * bb$
  vx[1] = (-50 * v[1] + 96 * v[2] - 72 * v[3] + 32 * v[4] - 6 * v[5]) * bb$
  vx[2] = (-6 * v[1] - 20 * v[2] + 36 * v[3] - 12 * v[4] + 2 * v[5]) * bb$
  vx[n$-1] = (-2 * v[n$-4] + 12 * v[n$-3] - 36 * v[n$-2] + 20 * v[n$-1] + 6 * v[n$]) * bb$
  vx[n$] = (6 * v[n$-4] - 32 * v[n$-3] + 72 * v[n$-2] - 96 * v[n$-1] + 50 * v[n$]) * bb$
end      [SET bb$ = 1/(24 DX)]
```

The end-value assignments again overwrite part of the vector assignment.

The initial temperature $\mathbf{u}(\mathbf{x}, 0)$ has to be 8000 K for all x along the rod. At $\mathbf{x} = 0$, the rod is insulated, but at $\mathbf{x} = L$ it radiates according to a fourth-power law, so that we have mixed-type boundary conditions at the ends of the rod:

$$u_x = 0 \text{ (for } x = 0, \text{ all } t) \quad u_x = E[UA^4 - u(L)^4] \quad \text{(for } x = L, \text{ all } t)$$

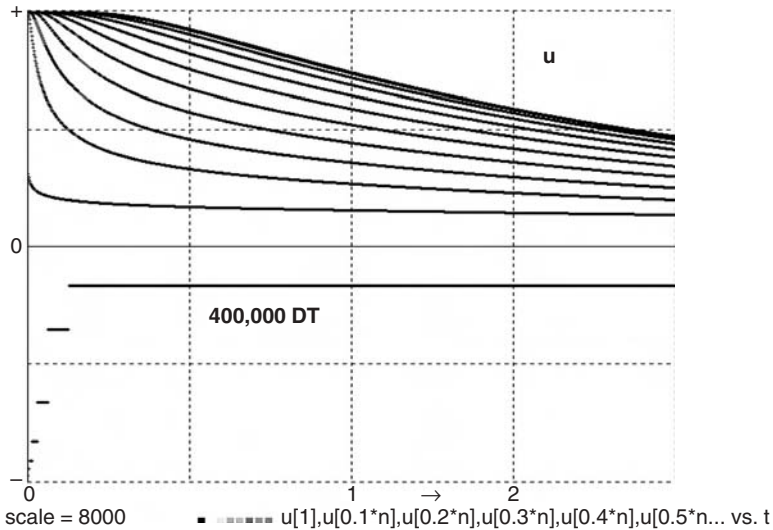
E and $UA^4 = UA^4$ are given constants. The experiment-protocol script sets the given initial state-variable values $u[i]$ with

```
for i = 1 to n | u[i] = scale | next
```

where the graph scale **scale** is set equal to the given initial temperature $u(\mathbf{x}, 0) = 8000$.

The DYNAMIC program segment in Figure 7-7 invokes **DDx** to produce the partial-derivative vector \mathbf{ux} with

```
invoke DDx(n, bb, u, ux)
```

HEAT-CONDUCTION PARTIAL DIFFERENTIAL EQUATION

```
ARRAY vx$[1], v$[1] | -- dummy arrays for SUBMODEL
```

```
--
```

```
-- Schiesser numerical-differentiation operator
```

```
--
```

```
SUBMODEL DDx(n$, bb$, v$, vx$)
```

```
  Vector vx$ = bb$ * (v$[1] - v$[-1])
```

```
  vx$[1] = bb$ * (-3 * v$[1] + 4 * v$[2] - v$[3])
```

```
  vx$[n] = bb$ * (3 * v$[n] - 4 * v$[n-1] + v$[n-2])
```

```
end
```

```
irule 15 | ERMAX = 0.001 | -- Gear-type integration
```

```
n = 51
```

```
STATE u[n] | ARRAY ux[n], uxx[n], U[n]
```

```
scale = 8000 | TMAX = 2 | NN = 1200
```

```
for i = 1 to n | u[i] = scale | next | -- initial conditions
```

```
L = 2 | UA = 400 | E = 1.73E-09 | UA4 = UA^4
```

```
--
```

```
DX = L/(n - 1) | bb = 1/(2 * DX)
```

```
DT0 = 0.0025 | DT = DT0/(n^2)
```

```
--
```

```
drun
```

```
DYNAMIC
```

```
invoke DDx(n, bb, u, ux) | -- differentiate u to get ux
```

```
ux[1] = 0 | ux[n] = E * (UA4 - u[n]^4) | -- boundary values
```

```
invoke DDx(n, bb, ux, uxx) | -- differentiate ux to get uxx
```

```
Vectr d/dt u = uxx
```

and then *overwrites the end values* **ux[1]** and **ux[n]** *to establish the given boundary values*

$$\mathbf{ux}[1] = 0 \quad | \quad \mathbf{ux}[n] = E * (\mathbf{UA}^4 - \mathbf{u}[n]^4)$$

DDx is invoked again to generate **uxx** with

$$\text{invoke DDx}(\mathbf{n}, \mathbf{bb}, \mathbf{ux}, \mathbf{uxx})$$

The given partial differential equation (7-2) is then programmed as

$$\mathbf{Vectr} \, d/dt \, \mathbf{u} = \mathbf{uxx} \quad (7-3)$$

This simple diffusion problem was solved using Gear-type integration with maximum relative error **ERMAX** = 0.001. An inexpensive 2.4-GHz personal computer produced the solution in less than 20 ms with the display turned off. Runtime compilation also took less than 20 ms. Interestingly, the solution at **x = L** for **n = 11** was within 0.2% of that for **n = 51**.

(c) Numerical Problems

The programming technique described in Section 7-11a is convenient, and the problem compiles as easily for **n = 200** as for **n = 10** if one does not run out of memory. But the solution accuracy must be critically reviewed in every case (see also Section 7-13).

First, derivative approximations involve small differences of larger numbers. With double-precision (64-bit) arithmetic, round-off errors are usually negligible. But note that the differential-equation system (7-3) implies differential-equation time constants of the order of **DX²**. Simple fixed-step integration rules then require integration steps **DT** of that order of magnitude [6], and the total number of operations would increase (for our case of one space dimension) with **n³**. MOL differential-equation systems can involve larger time constants as well (“stiff” system), so that the use of a variable-step implicit integration rule is indicated.

Figure 7-7 shows how the integration step **DT** changes. Our simple heat-conduction example was easy to solve because **u** changes rapidly only for small **x**, and then only initially. But it is not always so easy to obtain a stable solution. References [6,7] show more examples.



FIGURE 7-7. Solution and program for the heat-conduction or diffusion equation (see text). Display commands are not shown.

7-12. The Heat-conduction Equation in Cylindrical Coordinates

The partial differential equation

$$u_t = u_{xx} + u/x \quad (7-4)$$

models axially symmetric heat conduction in an infinitely long cylinder of radius R , using cylindrical coordinates with radius x [6]. We shall solve the partial differential equation (7-4) for the temperature $u(x, t)$, given the initial temperature $u(x, 0) = 0$ and the fixed boundary temperature $u(R, t) = u_0$. We need no experiment-protocol loop for the initial $u[i] = 0$, since all array components default to 0. But on the cylinder axis ($x = 0$), we must program the boundary condition $u_x(0, t) = 0$ for all values of t .

Figure 7-8 shows the complete program. The experiment-protocol script defines n radius values $x[1] = 0, x[2], \dots, x[n] = R$ of the radius x with

$$DX = R/(n-1) \quad \text{and} \quad \text{for } i = 1 \text{ to } n \quad | \quad x[i] = (i - 1) * DX \quad | \quad \text{next} \quad (7-5)$$

The DYNAMIC program segment derives the vectors u_x and u_{xx} and sets the given boundary conditions for $u[n] = u_0$ and $u_x[1] = 0$ by overwriting $u[1]$ and $u[n]$ as in Section 7-11. The partial differential equation (7-4) is then represented by

$$\text{Vectr } d/dt \, u = u_{xx} + u/x$$

To avoid division by 0, we overwrite the differential equation for $u[1]$ with the correct differential equation

$$d/dt \, u[1] = 2 * u_{xx}[1]$$

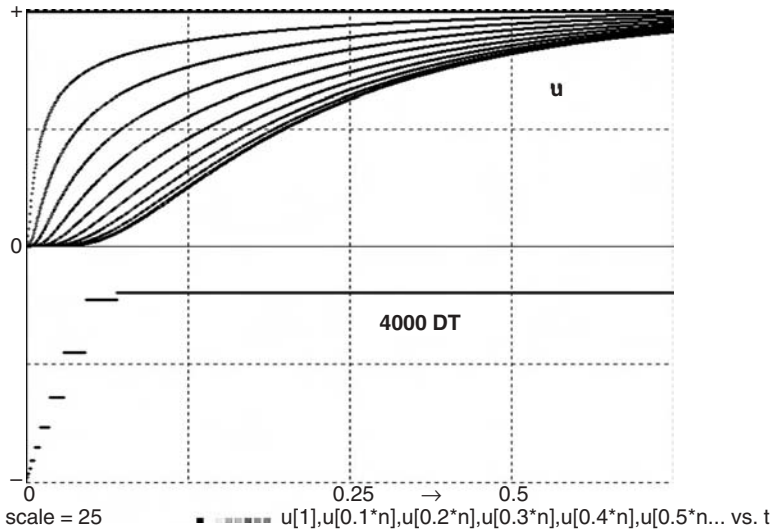
derived by l'Hôpital's rule ($u/x \rightarrow u_{xx}/1$ as $x \rightarrow 0$). Figure 7-8 shows solution time histories of the temperature u for the n uniformly spaced radius values defined by Eq. (7-5).

7-13. Generalizations

Our space-derivative submodels apply to many systems of parabolic linear or nonlinear partial differential equations with one space dimension, but problems



FIGURE 7-8. Solution of the partial differential equation for radial heat conduction in an infinite cylinder. Display commands are not shown.



-- HEAT-CONDUCTION PARTIAL DIFFERENTIAL EQUATION
 -- radial heat conduction, in cylindrical coordinates; x is radius

 ARRAY vx[1], v[1] | -- dummy arrays for SUBMODEL

--

SUBMODEL DDx(n\$, bb\$, v, vx)

Vector vx = bb\$ * (v{1} - v{-1})

vx[1] = bb\$ * (-3 * v[1] + 4 * v[2] - v[3])

vx[n\$] = bb\$ * (3 * v[n] - 4 * v[n-1] + v[n-2])

end

 irule 15 | ERMAX = 0.001 | -- Gear-type integration

n = 51

STATE u[n] | ARRAY x[n], ux[n], uxx[n]

 R = 1 | u0 = 25 | -- cylinder radius, surface temperature
 scale = u0

--

DX = R/(n - 1) | bb = 1/(2 * DX)

DT0 = 0.00025 | DT = DT0/(n^2)

TMAX = 0.5 | NN = 1000

--

for i = 1 to n | x[i] = (i-1) * DX | next

-- initial conditions u[i] = 0 need not be programmed

drun

 DYNAMIC

u[n] = u0 | -- set boundary value at x = R for each t

invoke DDx(n, bb, u, ux) | -- differentiate u to get ux

ux[1] = 0 | -- set boundary value for x = 0

invoke DDx(n, bb, ux, uxx) | -- differentiate ux to get uxx

Vectr d/dt u = uxx + ux/x | -- note: not valid for x = 0

d/dt u[1] = 2 * uxx[1] | -- from l'Hopital's rule

of order higher than 1 in the time dimension often run into serious numerical-stability problems. Schiesser and Silebi successfully solved the advection equation [6]

$$u_t = -Vu_x$$

and the related partial differential equation

$$u_t = -Vu_x + a(U - u)$$

which models a simple heat exchanger with flow velocity \mathbf{V} (Section 7-14) [7]. But direct application of the space-derivative submodels in Table 7-1 to the wave equation

$$u_{tt} = au_{xx}$$

fails. Schiesser [6] solved this problem by deriving a second-derivative operator for computing u_{xx} in one step, but boundary-condition setting becomes more complicated.

For problems with more than one space dimension, the vector compiler produces ordinary differential equations just as easily, for one can treat, for example, two-dimensional arrays as equivalent vectors (Section 3-11). But assignment of multidimensional boundary values is likely to be cumbersome, just as it would be in a Fortran program. An intelligent choice of the coordinate system can simplify model and program, as in the example of Figure 7-12.

7-14. A Simple Heat-exchanger Model

In the simple heat-exchanger model programmed in Figure 7-9a, $u(\mathbf{x}, t)$ is the temperature of a fluid running through a tube between $\mathbf{x} = 0$ and $\mathbf{x} = \mathbf{L}$. The initial fluid temperature is $u = T_0c = 0$. The fluid transfers heat to or from a surrounding constant-temperature annulus; longitudinal heat conduction is neglected. An initial step change T_0c in the inlet temperature at $\mathbf{x} = 0$ then travels down the tube by advection. $u(\mathbf{x}, t)$ satisfies the partial differential equation

$$u_t = -Vu_x + a(U - u)$$

where \mathbf{V} is the constant fluid velocity, and \mathbf{U} the annulus temperature, which is assumed to be constant here. The program in Figure 7-9 solves this partial differential equation using the second-order Schiesser differentiation operator as in Section 7-11.

-- SIMPLE HEAT EXCHANGER (Schiesser and Silebi, 1997)

display N1 | display C8 | display R

Schiesser numerical-differentiation operator

```
--
ARRAY vx[1], v[1] | --                               dummy arrays for submodel
SUBMODEL DDx(n$, bb$, v, vx)
  Vector vx = (v{1} - v{-1}) * bb$
  vx[1] = (-3 * v[1] + 4 * v[2] - v[3]) * bb$
  vx[n$] = (3 * v[n$] - 4 * v[n$ - 1] + v[n$ - 2]) * bb$
end
```

```
L = 100 | --                                           heat-exchanger tube length
V = 10 | --                                           flow velocity
rho = 1 | --                                           density of fluid in tube
CP = 1 | --                                           specific heat of fluid
D = 2 | --                                           tube diameter
H = 0.1 | --                                           heat-transfer coefficient
a = 4 * H/(rho * CP * D)
```

```
--
Tac = 100 | --                                         constant annulus temperature
T0c = 0 | --                                         initial temperature in tube
Tec = 50 | --                                         tube-entry temperature
```

```
n = 201 | STATE u[n] | ARRAY ux[n]
```

```
DX = L/(n-1) | bb = 0.5/DX
DT = 0.0005 | TMAX = 10 | NN = 20000 | scale = 100
irule 4 | — RK4 rule
```

```
X = 0.5 * (n-1) * DX/V | --
                                         time delay used in theoretical solution
```

```
--
                                         program initial conditions u[k]
for k = 1 to n | u[k] = T0c | next
U = Tac | —                                         constant annulus temperature
drun | write "n = ";n
```

DYNAMIC

```
u[1] = Tec | —                                         set tube-entry boundary value for u
invoke DDx(n, bb, u, ux) | --                         differentiate u to get ux
Vectr d/dt u = -V * ux + a * (U - u)
```

```
--
                                         compare u with theoretical solution f
```

```
f = 2 * (Tac + (T0c - Tac * exp(-a * t) * (1 - swtch(t - X))
                                         + swtch(t - X) * ((Tec - Tac) * exp(-a*X))) - scale
uu = 2 * u[0.5*(n-1)] - scale | errx5 = 2.5 * (uu - f) - 0.5 * scale
dispt uu, errx5, f
```

FIGURE 7-9a. Program for computer simulation of a simple heat exchanger.

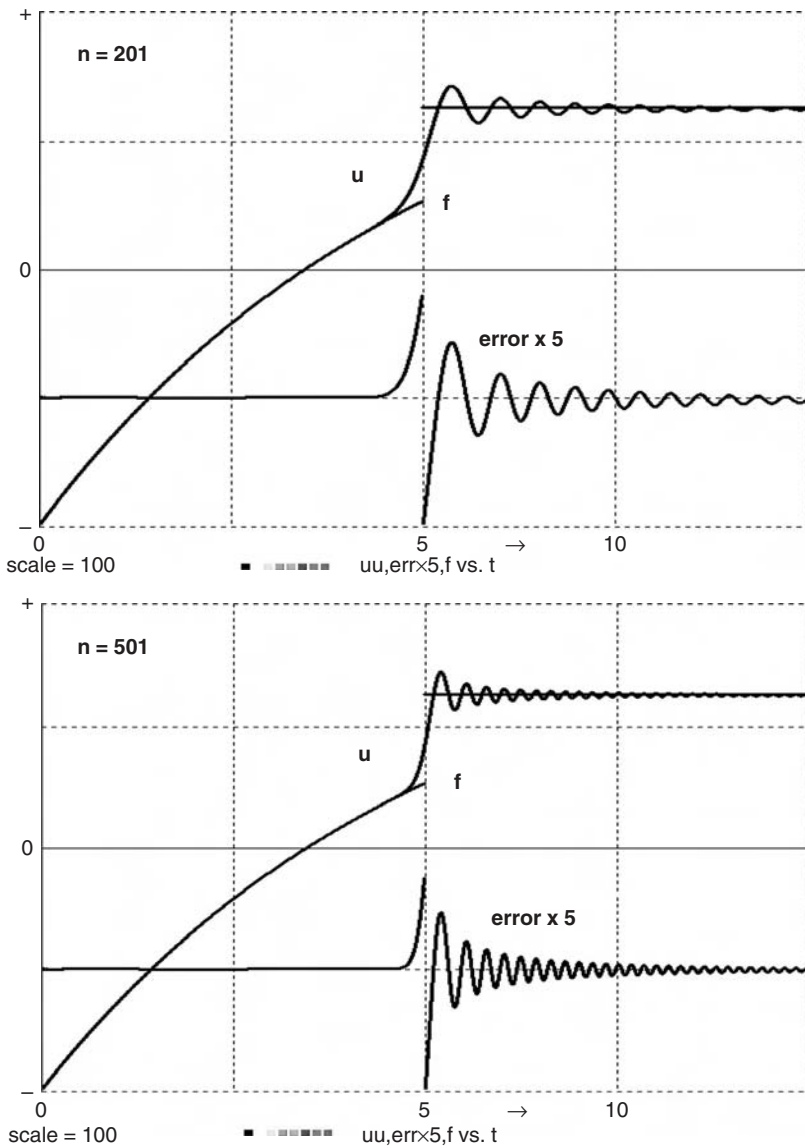


FIGURE 7-9b. Theoretical and computed time histories of the heat-exchanger tube temperature $u(x, t)$ for $x = L/2$.

For each value of x along the tube, the theoretical solution in Figure 7-9 combines an exponential function increasing from $u(x, t) = 0$ to $u(x, t) = U$ with a step-function temperature change due to advection of the inlet temperature step. Figure 7-9b compares this theoretical solution with the numerical solution. The oscillatory response shown for $n = 201$ and 501

is typical of numerical solutions of hyperbolic partial differential equations [6,7].

REPLICATION OF AGROECOLOGICAL MODELS ON MAP GRIDS

7-15. A Geographical Information System

The SAMT (Spatial Analysis and Modeling Tool) program package developed by R. Wieland [10,11] is a simple geographical information system for description and manipulation of ecological data. SAMT declares and stores arrays of numerical landscape-feature values for a specified grid of geographical locations. Examples of landscape features are

- geographical coordinates for each grid point (x, y , altitude)
- physical data such as temperature, soil moisture, and species counts at each grid point.

SAMT can assign and calculate functions that relate different landscape features at any one grid point, say

$$q_1 = q_2 + q_3 \quad q_1 = \cos(q_2) \quad q_1 = \text{calc}(q_2, q_3, \dots)$$

Functions such as **calc**(q_2, q_3, \dots) are either numerical expressions or regression functions previously created by simple neural-network or fuzzy-set models [11].

SAMT can also assign and store grid-point data values that depend on data at other grid points, such as the distance of the current grid point from another grid point, say, from a city or from a bird's nest; or the shortest distance to a river or road. SAMT can, moreover, accumulate statistics such as averages and statistical relative frequencies for an entire set of grid points. Last but not least, SAMT can draw maps showing grid-point data values in different colors, or showing contour lines for different landscape features (Fig. 7-10).

7-16. Modeling the Evolution of Landscape Features

The original SAMT database described a landscape at coarsely spaced sampling times t (e.g., once per day, per month, and per year), but the program did not relate landscape features at different sampling times. In contrast, DESIRE uses small time steps Δt to simulate continuous changes. A combination of SAMT and DESIRE can model changes of landscape features at each separate grid point with differential equations and/or difference equations. SAMT/DESIRE runs under both Linux and Windows (Figs 7-11 and 7-12).

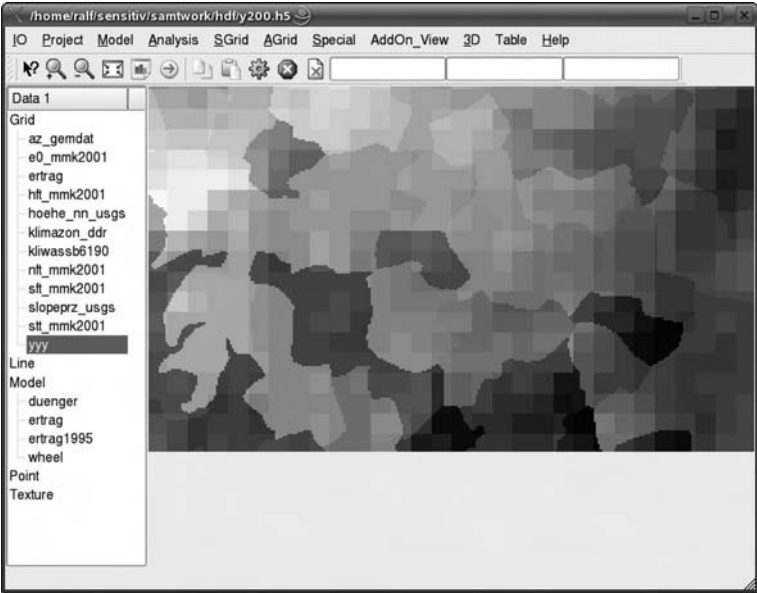


FIGURE 7-10. A map of relative vegetation density produced by the SAMT geographical information system [12]. The original display was in color.

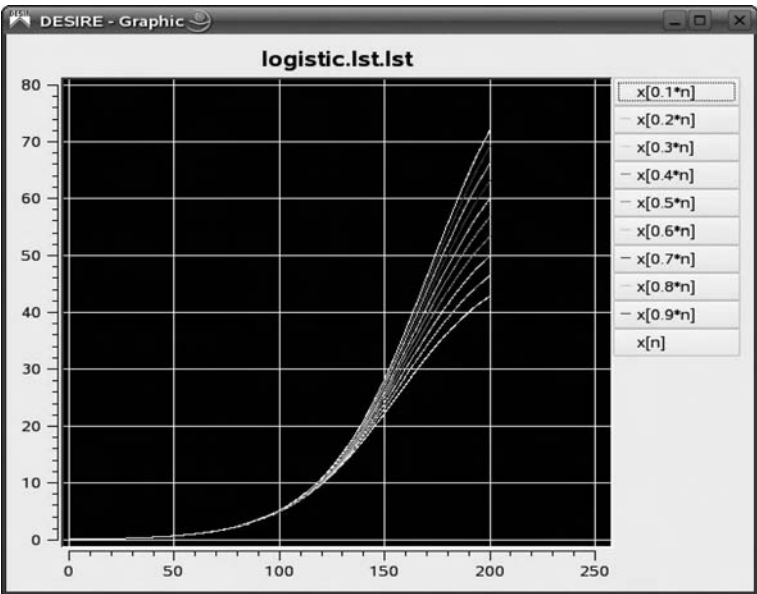


FIGURE 7-11. A self-scaling SAMT/DESIRE graphics window. Three-dimensional graphs can also be displayed.

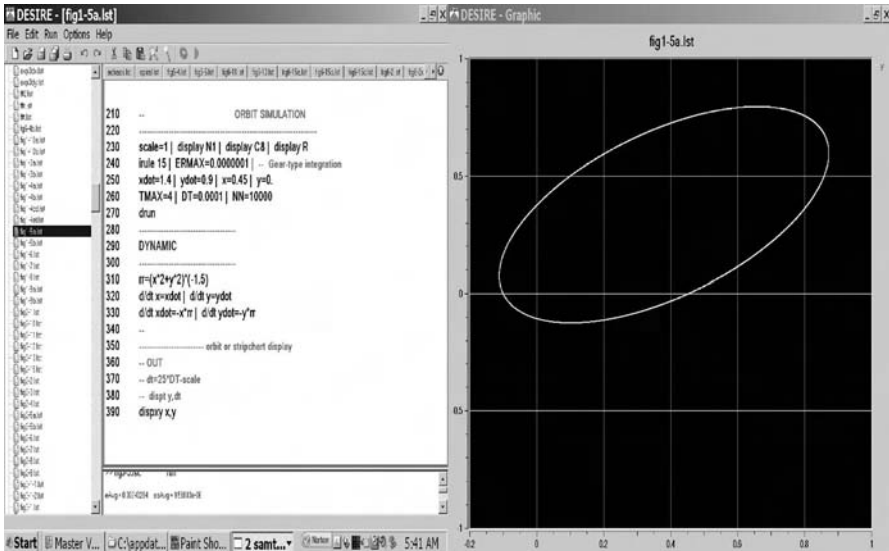


FIGURE 7-12. SAMT/DESIRE dual-monitor display under Microsoft Windows™. To simplify interactive modeling for nontechnical users, SAMT/DESIRE simulations are programmed and run from a special editor window designed by R. Wieland and X. Holtmann [12]. A clickable program-selection directory is on the left.

Some landscape features will be state variables with specified initial values. Other landscape features, such as intermediate results and data transferred to and from the SAMT data base, are defined variables, as described in Sections 1-2 and 2-1. One might, for example, have a differential-equation system modeling competition between local predator and prey populations (Section 1-12) at each grid point. Local crop growth is another promising application [15].

REFERENCES

1. F. Breitenacker and I. Husinsky, Results of the EUROSIM comparison "Lithium Cluster Dynamics," *Proceedings of EUROSIM*, Vienna, 1995.
2. G. A. Korn, *Interactive Dynamic System Simulation with Microsoft Windows*, Francis and Taylor, London, 1998.
3. B. Kosko, *Neural Networks and Fuzzy Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
4. G. A. Korn, *Neural Networks and Fuzzy-logic Control on Personal Computers and Workstations*, MIT Press, Cambridge, MA, 1995.

5. G. A. Korn, Simulating a fuzzy-logic-controlled nonlinear servomechanism, *SAMS*, **34**, 1999, pp. 35–52.
6. W. E. Schiesser, *The Numerical Method of Lines*, Academic Press, New York, 1991.
7. W. E. Schiesser and C. A. Silebi, *Computational Transport Phenomena*, Cambridge University Press, Cambridge, 1997.
8. A. Tveito and R. Winther, *Introduction to Partial Differential Equations*, Springer, New York, 1998.
9. G. A. Korn, Interactive solution of partial differential equations by the method of lines, *Math and Computers in Simulation*, **1644**, 1999, 1–10.
10. G. A. Korn, Using a runtime simulation-language compiler to solve partial differential equations by the method of lines, *SAMS*, **37**, 2000, pp. 141–149.
11. R. Wieland, Spatial Analysis and Modeling Tool, Report, ZALF Institut für Landschaftssystemanalyse, Muencheberg, Germany.
12. G. A. Korn and X. Holtmann, Spatial analysis and modeling tool – a new software package for landscape analysis, *Simulation News Europe*, December 2005.
13. G. A. Korn, Simplified function generators based on fuzzy-logic interpolation, *Simulation Practice and Theory*, **7**, 2000, pp. 709–717.
14. M. Wang and J. M. Mendel, Fuzzy basis functions, *IEEE Trans. Neural Networks*, **3**, 1992, pp. 807–818.
15. R. Wieland and M. Voss, Spatial analysis and modeling tool — possibilities and applications, *Proc. IAST-ED Conference on Environmental Modeling and Simulation*, November 2004.

Appendix

ADDITIONAL REFERENCE MATERIAL

A-1. Example of a Radial-basis-function Network

Figure A-1 shows a complete program for a radial-basis-function network learning the target function $Y(\mathbf{x}) = 0.95 * \sin(\mathbf{x}[1]) * \cos(\mathbf{x}[2]) * (\mathbf{x}[3] - 0.5)$. The input-pattern dimension is $\mathbf{nx} = 3$. Like all other example programs, this program is in the book CD. One can easily try different target functions $Y(\mathbf{x})$, different numbers \mathbf{n} of radial-basis centers, and different values of the Gaussian-bump spread parameter \mathbf{a} .

The program incorporates several of the programming tricks introduced in Chapter 6. The experiment-protocol script calls two separate DYNAMIC program segments. The DYNAMIC segment labeled **COMPETE** runs first. This implements a competitive-layer network¹ that finds an $\mathbf{n} \times \mathbf{nx}$ template matrix \mathbf{P} whose \mathbf{n} rows represent cluster centers for uniformly distributed input vectors **Vector** $\mathbf{x} = 1.5 * \text{ran}()$ (Sections 6-14 to 6-16). We then use these cluster centers as radial-basis centers. Different input distributions can be substituted if desired.

¹ **crit = 0** specifies the Ahalt-type conscience algorithm (Section 6-16b).

-- RADIAL-BASIS-FUNCTION NETWORK

```

-----
nx = 3 | ARRAY x[nx] | --input
n = 300 | --                               number of radial-basis centers
ARRAY P[n,nx], v[n], h[n], z[nx] | --       for competition
ARRAY ff[n] + ff0[1] = f | ff0[1] = 1 | --   f has bias component
ARRAY pp[n]
--                                           weights include biases
ARRAY W[1,n+1], y[1], error[1]
-----

lratex = 0.2 | lratex = 0.3 | kappa = 0.9998 | lrate0 = 0 | a=4
crit = 0 | --                               crit = 0 for FSCL conscience
NN = 100000 | --                             number of trials for template generation
-----

--                                           learn radial-basis centers
drun COMPETE
write "type go to continue" | STOP
-----

---                                           template matrix P generates squared radii
for k = 1 to n
  pp[k] = 0
  for j = 1 to nx | pp[k] = pp[k] + P[k,j]^2 | next
next
-----

--                                           now train radial-basis-function network
NN = 100000 | --                             number of training trials
drun
write "type go for a recall run" | STOP
NN = 2000 | lratex = 0 | lrate0 = 0 | --       recall run
drun
-----

DYNAMIC
-----

lratex = kappa * lratex + lrate0 | --         reduce learning rate
--
Vector x = ran() | --                         training pairs
Y=sin(x[1])*cos(x[2])*(x[3] - 0.5) | --       or use other functions
--
--                                           xx - 2 * P * x + pp is the vector of squared radii
DOT xx = x * x | Vector ff = exp(a * (2 * P * x - xx - pp))
--
Vector y = W * f | --                         radial-basis-function expansion

```

FIGURE A-1a. Complete program for a three-dimensional radial-basis-function network learning the three-input function $Y(x) = 0.95 * \sin(x[1]) * \cos(x[2]) * (x[3] - 0.5)$. Other error displays could be used. This program employs the Casasent algorithm described in Section 6-13b.

```

Vector error = Y - y | --                                LMS algorithm
DELTA W = lratex * error * f
-----
ERRORx10 = 10 * abs(error[1]) - scale | y1 = y[1]        stripchart display
dispxy Y, y1, ERRORx10, y1
-----
label COMPETE
lratex = kappa * lratex + lrate0 | --                    reduce learning rate
Vector x = 1.5 * ran()
CLEARN v = P(x) lratex, crit | --                       compete to get template
Vectr delta h = v | --                                  conscience counter
Vector z = P% * v
dispxy z[1], z[2] | --                                   shows 2 dimensions only

```

FIGURE A-1a. (Continued)

The main DYNAMIC program segment represents the radial-basis-function network proper. We use the efficient Casasent-type algorithm of Section 6-13b to compute the distances between the current input vector \mathbf{x} and the $\mathbf{n} = 300$ radial-basis centers. This allows us to compute the desired vector \mathbf{f} of \mathbf{n} radial-basis functions $\mathbf{f}[\mathbf{k}]$. The LMS algorithm then produces the optimal connection-weight matrix \mathbf{W} for the radial-basis-function expansion

Vector $\mathbf{y} = \mathbf{W} * \mathbf{f}$

Note that our expansion includes a bias term implied by the array declaration

ARRAY ff[n] + ff0[1] = f | ff0[1] = 1

as described in Footnote 3, Chapter 6.

A-2. A Fuzzy-basis-function Network

Figure A-2a represents a fuzzy-basis-function network learning $\mathbf{Y}(\mathbf{x}) = 2 * \sin(0.5 * \mathbf{x}[1]) * \cos(0.1 * \mathbf{x}[2])$. The input-pattern dimension is $\mathbf{nx} = 2$. We use products of the triangular membership functions defined by the submodel in Section 7-7b as basis functions.

There are $\mathbf{N1}$ membership functions $\mathbf{mb1}$ for $\mathbf{x}[1]$ peaking at $\mathbf{x}[1] = \mathbf{X1}[1]$, $\mathbf{X1}[2]$, ..., $\mathbf{X1}[\mathbf{N1}]$, and $\mathbf{N2}$ membership functions $\mathbf{mb2}$ for $\mathbf{x}[2]$ peaking at $\mathbf{x}[2] = \mathbf{X2}[1]$, $\mathbf{X2}[2]$, ..., $\mathbf{X2}[\mathbf{N2}]$, and therefore $\mathbf{n} = \mathbf{N1N2}$ fuzzy-basis centers with coordinates $\mathbf{X1}[\mathbf{i}]$, $\mathbf{X2}[\mathbf{k}]$. Simple **data/read** assignments allow one to quickly enter fuzzy-basis-center coordinates $\mathbf{X1}[1]$, $\mathbf{X2}[1]$, ..., $\mathbf{X1}[\mathbf{N1}]$ and $\mathbf{X2}[1]$, $\mathbf{X2}[2]$, ..., $\mathbf{X2}[\mathbf{N2}]$ in ascending order. This enables one to try different

```

--
FUZZY-BASIS-FUNCTION NETWORK
-----
FUNCTION Y(p$, q$) = 2 * sin(0.5 * p$) * cos(0.1 * q$)
-----
--
triangle membership functions
--
ARRAY X$[1], mb$[1] | --
SUBMODEL fuzzmemb(N$, X$, mb$, input$)
Vector mb$ = SAT((X$ - input$)/(X$ - X${1}))
Mbb = mb$[1] | mcc = mb$[N$ - 1]
Vector mb$ = mb${-1} - mb$
mb$[1] = 1 - mbb | mb$[N$] = mcc
end
-----
ARRAY x[2]
N1 = 7 | N2 = 5 | n = N1 * N2
ARRAY X1[N1], X2[N2]
ARRAY mb1[N1], mb2[N2], F[N1,N2] = f
ARRAY W[1,n], y[1], error[1]
-----
lratef = 0.2 | kappa = 0.9999 | lrate0 = 0.0
NN=20000
-----
data -0.9, -0.5, -0.1, 0, 0.1, 0.5, 0.9 | read X1
data -0.9, -0.5, 0, 0.5, 0.9 | read X2
-----
drun | -- function-learning run
write "type go for a recall run" | STOP
lratex=0 | lratef=0 | lrate0=0 | NN=2000 | display R
drun | -- recall run
-----
DYNAMIC
-----
-- lratef = kappa*lratef+lrate0 | -- reduce learning rate
--
Vector x = ran() | Y = TGT(x[1], x[2]) | -- training pairs
invoke fuzzmemb(N1, X1, mb1, x[1])
invoke fuzzmemb(N2, X2, mb2, x[2])
MATRIX F = mb1 * mb2 | -- joint membership functions
Vector y = W * f | Vector error = Y - y
DELTA W = lratef * error * f | -- LMS algorithm for output y
--
-----
stripchart display
ERRORx50 = 50 * abs(error[1]) - scale
x1 = x[1] | x2 = x[2] | -- shorten names to fit display list!
m2 = 0.5 * (0.5 * mb1[2] + scale)
m3 = 0.5 * (0.5 * mb1[3] + scale)
M2 = 0.25 * mb2[2] | M3 = 0.25 * mb2[3]
DISPX Y, y[1], Y, ERRORx50, x1, m2, x1, m3, x2, M2, x2, M3

```

FIGURE A-2a. Program for a two-dimensional fuzzy-basis-function network learning the two-input function $2 * \sin(0.5 * x[1]) * \cos(0.1 * x[2])$.

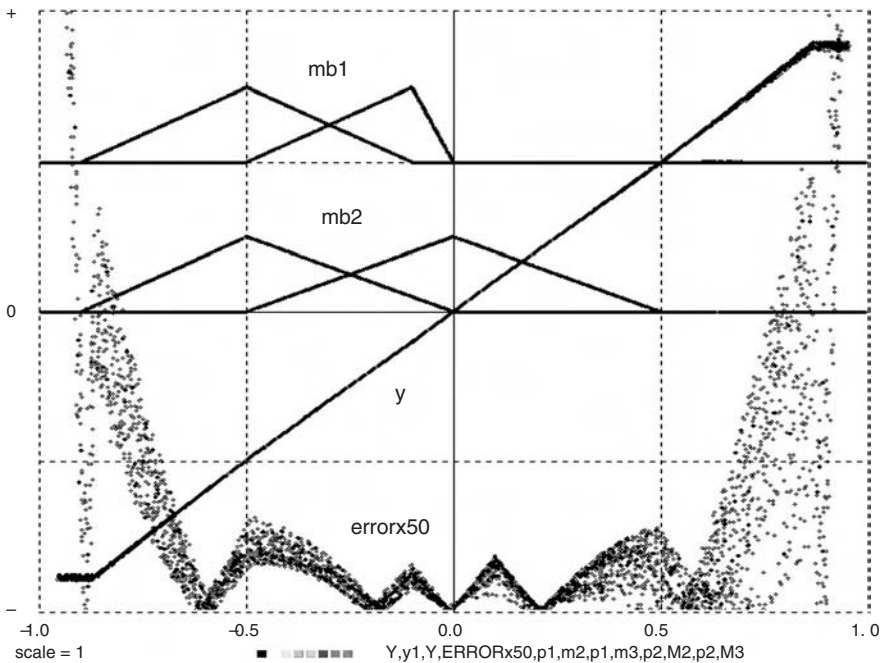


FIGURE A-2b. Display obtained with the fuzzy-basis-function network of Fig. A-2a. **y** and error are plotted against the target input **Y**. **mb1** and **mb2** are plotted against their respective arguments **x[1]** and **x[2]**. Note that different fuzzy basis functions were used for **x[1]** and **x[2]**.

basis-center locations by trial and error; it should be noted that basis-function spreads in the **x[1]** and **x[2]** directions adjust automatically when the fuzzy-basis-center locations are changed.

The results are shown in Figure A-2b. As in Section 7-7c, **sigmoid** (**a * q**) can be substituted for **SAT(q)** to try continuous basis functions.

A-3. The CLEARN Algorithm²

From Section 6-17, in

CLEARN **v = W(x) lrate, crit**

crit > 0 associates each template **W⁽ⁱ⁾** with two internal status flags, **committed[i]** and **badmatch[i]**, both initially set to 0 (see also the DESIRE Reference

² The new **CLEARN** algorithm described here replaces the now obsolete algorithm described in Reference [3].

Manual) [1,2]. We start with random values for n templates. Each learning step presents an input pattern \mathbf{x} and finds the best-matching template pattern $\mathbf{W}^{(l)}$ by least-squares competition. Then

- If the best-matching template satisfies the resonance criterion $\|\mathbf{x} - \mathbf{W}^{(l)}\| < \text{crit}$, update it and mark it with **committed**[l] = 1. This template pattern then no longer competes for less-close matches (see below).
- For $\|\mathbf{x} - \mathbf{W}^{(l)}\| > \text{crit}$, update $\mathbf{W}^{(l)}$ if it is uncommitted (**committed**[l] = 0). If $\mathbf{W}^{(l)}$ has been previously committed (**committed**[l] = 1), do not update $\mathbf{W}^{(l)}$ but reset the current search (**badmatch**[l] = 1).

In other words, **committed**[i] = 0 flags identify as yet uncommitted templates that are allowed to acquire new patterns with or without resonance. The process terminates when all n templates are committed. As in classical ART, a fast-learn mode for noise-free patterns can simply set $\mathbf{W}^{(l)} = \mathbf{x}$ instead of gradual updating.

REFERENCES

1. G. A. Korn, *Neural Networks and Fuzzy-logic Control on Personal Computers and Workstations*, MIT Press, Cambridge, MA, 1995.
2. G. A. Korn, New, faster algorithms for supervised competitive learning: counterpropagation and adaptive-resonance functionality, *Neural Processing Letters*, **9**, 1999, pp. 107–117.
3. M. Wang and J. M. Mendel, Fuzzy basis functions, *IEEE Trans. Neural Networks*, **3**, 1992, pp. 807–818.

TABLE A-1. DESIRE Integration Rules

(a) <i>Euler and Runge-Kutta Rules</i> (up to 40,000 state variables)
k1 = G(x, t) * DT
irule 1 (fixed-step second-order Runge-Kutta-Heun—this is the default rule)
k2 = G(x + k1, t + DT) * DT
x = x + (k1 + k2)/2
irule 2 (explicit Euler rule, first order)
Users may change DT as a function of t .
x = x + G(x, t) * DT = x + k1
irule 3 (fourth-order Runge-Kutta)
Users may change DT in the course of a simulation run.
k2 = G(x + k1/2, t + DT/2) * DT k4 = G(x + k3, t + DT) * DT
k3 = G(x + k2/2, t + DT/2) * DT
x = x + (k1 + 2 * k2 + 2 * k3 + k4)/6
Variable-step Runge-Kutta rules compare two Runge-Kutta formulas of different order. The step size doubles when the absolute difference is less than ERMIN , until DT reaches DTMAX . If the system variable CHECKN is a positive integer n , then the step size DT is halved if the absolute difference of the two expressions for the n th state variable exceeds ERMAX . If CHECKN = 0 , then DT is halved when the relative difference exceeds ERMAX for any state variable. A variable-step deadlock error results if DT attempts to go below DTMIN ; the deadlocked absolute difference can then be read in ERRMAX .
irule 4 (variable-step Runge-Kutta 4/2) compares the fourth-order Runge-Kutta result with x = x + k2
irule 5 (second-order Runge-Kutta-Heun) is similar to irule 1 , but users may change DT during a run.
irule 6 is a spare, not currently implemented.
irule 7 (variable-step Runge-Kutta 2/1) compares
k2 = G(x + k1, t + DT)
x = x + (k1 + k2)/2 with x = x + k1
irule 8 (variable-step Runge-Kutta-Niesse) compares
k2 = G(x + k1/2, t + DT/2) * DT
k3 = G(x - k1 + 2 * k2, t + DT) * DT
x = x + (k1 + 4 * k2 + k3)/6 with x = x + (k1 + k3)/2
(b) <i>Adams-Type Variable-Order/Variable-Step Rules</i> (up to 600 state variables)
irule 9 functional iteration
irule 10 chord/user-furnished Jacobian
irule 11 chord/differenced Jacobian
irule 12 chord/diagonal Jacobian approximation

Table A-1. (*Continued*)

(c) Gear-Type Variable-Order/Variable-Step Rules (for stiff systems, up to 600 state variables) [1,2]
--

irule 13 functional iteration
irule 14 chord/user-furnished Jacobian
irule 15 chord/differenced Jacobian
irule 16 chord/diagonal Jacobian approximation

Rules 9 to 16 employ a user-specified maximum relative error **ERMAX**, which must be specified in the interpreter program for all state variables; values equal to 0 are automatically replaced by 1 (see examples **orbitx.lst**, **to22x.lst**, and **rule15.lst**). The initial value of **DT** must be set low enough to prevent integration-step lockup.

irule 10 and **irule 4** need a user-furnished $n \times n$ Jacobian matrix for n state variables, say **J** (see the *DESIRE* reference manual).

REFERENCES

1. C. W. Gear, DIFSUB, Algorithm 407, *Communications ACM*, **14**, No. 3, 1971.
2. A. C. Hindmarsh, LSODE and LSODI, *ACM/SIGNUM Newsletter*, **15**, No. 4, 1980.

TABLE A-2. DESIRE Fast Fourier Transforms

1. FFT F, NN, x, y implements the discrete Fourier transform

$$x[i] + j y[i] \leftarrow \sum_{k=1}^{NN} (x[k] + j y[k]) \exp (-2\pi j i k / NN) \quad (i = 1, 2, \dots, NN)$$

FFT I, NN, x, y implements the discrete inverse Fourier transform

$$x[k] + j y[k] \leftarrow \frac{1}{NN} \sum_{i=1}^{NN} (x[i] + j y[i]) \exp (2 \pi j i k / NN) \quad (k = 1, 2, \dots, NN)$$

2. When the $x[k]$, $y[k]$ represent NN time-history samples taken at the sampling times

$t = 0, \text{COMINT}, 2 \text{COMINT}, \dots, \text{TMAX}$ with $\text{COMINT} = \text{TMAX}/(NN - 1)$

then the time-domain period associated with the discrete Fourier transform equals

$$T = NN * \text{COMINT} = NN * \text{TMAX}/(NN - 1)$$

(not TMAX). Approximate frequency-domain sample values of the ordinary integral Fourier transform are represented by $x[i] * T/NN$, $y[i] * T/NN$.

3. If the $x[i]$, $y[i]$ represent NN frequency-domain samples taken at the sample frequencies

$f = 0, \text{COMINT}, 2 \text{COMINT}, \dots, \text{TMAX}$ with $\text{COMINT} = \text{TMAX}/(NN - 1)$

then

t represents f (frequency)

COMINT represents $1/T$ (frequency-domain sampling interval)

TMAX represents $(NN - 1)/T$

$NN * \text{TMAX}/(NN - 1)$ represents NN/T (frequency-domain "period")

PROGRAMS IN THE BOOK CD

The CD furnished with this book contains the regular open-software distribution of OPEN DESIRE, including executable programs for personal computers, source code, a comprehensive reference manual, and many examples. This software can be freely used, modified, and redistributed under the terms of the Free Software Foundation's General Public License (GPL), a copy of which is part of the distribution. The GPL states explicitly that there is no warranty of any kind.

We have included *OPEN DESIRE for Linux*, which can be recompiled for other Unix-type operating systems, including Cygwin (Unix under Microsoft WindowsTM). The Linux program is the most technically advanced version of DESIRE, uses Xwindows graphics, and handles up to 40,000 first-order differential equations.

OPEN DESIRE for Microsoft WindowsTM is a smaller educational version, also written in C, and similar to the commercially available Windows/2000 program.³ It uses MS-DOS fullscreen graphics, and handles up to 20,000 differential equations. The educational program is available without charge by emailing the author at gatmkom@aol.com.

OPEN DESIRE includes over 100 example programs formatted for Linux and Cygwin (**.lst** files). The file **userexamples.doc** lists some of the titles. For the reader's convenience, examples referenced in the text were relabeled with figure numbers (e.g. **fig2-9.lst**).

Please refer to the Web site members.aol.com/gatmkorn for revision and new developments.

STREAMLINED OPERATION OF DESIRE PROJECTS UNDER LINUX

Referring to the DESIRE reference manual and the **README** file on the book CD, the OPEN DESIRE distribution files are all initially installed in a single installation directory, say **/desire**. User programs (**.lst** files) can be moved or written to any directory on the hard disk, say to a user's own project directory **/projects/mynewproject**. Such project directories can also contain text files, screenshots, notes or spreadsheets with results, reports being prepared, shell scripts, or any other files.

To run DESIRE, one opens a terminal window, **cds** to the installation directory, and starts one of the DESIRE programs, say **desire64**, with

./desire64,

³ DESIRE/2000 uses assembly language, is twice as fast as OPEN DESIRE for Windows, and adds more convenient control-key and mouse operations.

./desire64 1 would automatically load the last-used user program. A user program in the installation directory **/desire**, say **servo4.lst**, is loaded with

rld 'servo4'

Loading user programs from another directory requires a longer command such as

rld '../projects/mynewproject' or
rld 'home/username/projects/mynewproject'

But with the shell script **mk_syspict.sh** installed in the user's **\bin** directory *there is no need to type long rld commands. Simply clicking on the **servo4.lst** icon in any directory automatically loads the user program into DESIRE.* Typing **erun** in the terminal window then runs the program immediately, and **ed** or **edit** opens the program in an editor window.

The script file **mk_syspict.sh** on the book CD refers specifically to the installation directory named **/desire**. It is easy to edit the script to substitute another installation directory.

Index

- Absolute value, 28, 47, 49–50, 143
- Abstractions, 71, 74, 127, 154
- ACSL™, 4, 12
- Adaptive resonance theory (ART), 151–153
- Additive noise, 108
- Advection, 194
- Agroecological models/simulations, 63, 197–199
- Algebraic applications, 3, 12, 33
- Amplitude, implications of, 57–58, 86, 108, 117, 180
- Analog
 - code, 79
 - noise, 108
 - variables, 36
- Analog-to-digital conversion, 38, 40
- ARRAY**, 60
- Array declarations, 59–60
- Associative memory, 132
- Automotive engineering, 3
- Average-return vectors, 110–111

- Backpropagation networks, 138–141, 155, 159
- Band-pass filter, 108
- Bandwidth, 117
- Basic, 5
- Berkeley Madonna, 4
- Bias neural network vectors, 127
- Billiard-ball simulation, 20–22
- Binary code, xiii n1
- Binary selectors, 138–139, 150–151, 155
- Binomial distribution, 98
- Block-diagram composition, 4
- Boundary conditions, 189

- C, 4, 63 n6
- Cartesian coordinates, 18, 25
- Casasent's algorithm, 145

- Central-difference derivative approximations, 188–189
- Central limit theorem, 88 n3
- Centroid defuzzification, 178
- Chaotic time series, 161
- CLEARN** algorithm, 149–151, 153–154, 205–206
- Coin-toss study, 106
- Color, significance of, 28, 57–58
- COMINT**, 11, 34–35, 46
- Comparators, characteristics of, 42, 44
- Compilers
 - equation-language, 4
 - vectorizing, 60–63, 83
- Compiling, loop-unrolling, 67
- Complex numbers, 4, 25
- Computer runs, 3
- Computer simulation, *see specific software programs*
 - applications, 1–2, 12–30
 - how a run works, 5–12
 - numerical integration, 10–11
 - purpose of, 30
 - sampling 5, 8–12
 - speed of, 2
 - studies, 3–4
- Computing-time loss, 47
- Connection-parameter adjustments, 125
- Connection weight, 126–128, 133, 139–140, 144, 154–155, 157
- Conscience algorithms, 148
- Continuous difference-equation state variable
 - characteristics of, 52
 - hysteresis-models, 53–56
 - maximum-minimum-value holding, 53
 - simple backlash models, 53–54
- Continuous differential-equation programs, 36
- Continuous noise, 107–108

- Continuous-system simulation models, 2
- Continuous variables, 64
- Controller design, fuzzy-set logic, 179
- Control signals, 57–58
- Control system(s)
 - errors, 119
 - examples of, *see* Control-system examples
 - linear, 74–75, 122–123
 - noisy, 116–119
 - simulation, *see* Control-system simulations
- Control-system examples, *see also* Sampled-data control systems
 - electrical servomechanism with motor field delay and saturation, 22–24
 - frequency response, 24–25
- Control-system simulations
 - fuzzy-logic function generators, 181
 - neural networks, 141
- Coordinate systems, 72
- Correlation(s)
 - coefficients, 90, 97
 - matching, 153–154
- Counterpropagation learning, 147, 155–156
- Crossplotting, 2, 86–87
- Cygwin
 - applications, xiii n1, 4, 7
 - signal-generator program, 57
 - simple backlash transfer, 54
- Damped harmonic oscillator, 12–15
- Damping coefficients, 25, 64, 81, 87, 120
- data/read** assignment, 134
- Data assignments
 - noise studies, 106
 - sampled, *see* Sampled-data assignments
- Deadspace, 44, 54
- Debugging, 4
- Declaration arguments, 75
- Defined-variable assignments
 - characteristics of, 3, 10, 29
 - multilayer neural networks, 140
 - scalar, 72, 74
 - sorting, 12
 - vector, 66
- Derivative approximations, 191
- Derivative calls, 96
- DESIRE/2000, 4
- Difference equation
 - matrix, 70–71
 - parameters in, neural networks, 162–163, 167–168
 - programs, *see* Difference-equation programs
- Difference-equation programs
 - combined with sampled-data operations, 35–36
 - mixed continuous/sampled-data system examples, 38–41
 - sampled-data variables, initializing and resetting, 38
 - simple, 34–37
- Differential-algebraic-equation (DAE) systems, 3
- Differential equation(s)
 - absolute value and, 49–50
 - code, 35–36
 - data-assignment models, 32–38
 - Duffing's, 15–16
 - limiters, 45, 49–56
 - with linear harmonic oscillator, 22–24
 - maximum/minimum selection, 49–50
 - models, xiii n1, 2–3
 - noisy, 122
 - nonlinear systems, 125 n1
 - parameters in neural networks, 162–163, 167–168
 - polar-coordinate, 18
 - programs, *see* Difference-equation programs
 - sampling, 8
 - state-variable, *see* Differential-equation state variable
 - submodels, 76–78
 - switch functions, 45, 51–56
 - vectors/vectorization, 61–62, 64–66, 83
 - Volterra–Lotka, 19–20
- Differential-equation state variable
 - continuous, 52–56
 - signal generators, 56–58
 - signal modulators and, 56–58
- Differentiation operators, 188–191
- Digital controllers
 - guided-torpedo missile simulation, 38–40\
 - PID, plant simulation with, 40–41
 - signal quantification, 50–51
 - simulated, 46
- Digital filters, 35
- Digital signal processing, 67
- Discontinuous functions, 45, 47
- Discrete-event simulation, 1
- Dispersion, 93
- Display
 - colors, *see* Color
 - problems, 45–46
 - scale/scaling, 28, 53
- DOT** operations, 67, 96
- drunr** statements, 3–5, 19, 38, 52, 62, 71, 90
- Duffing's differential-equation system, 15–16
- Dummy arrays/variables, 75, 77–78
- DYNAMIC**, 14

- DYNAMIC-segment variables, 5, 8–10
- Dynamic-system simulation
 - computer modeling, 1–2
 - control system examples, 22–29
 - defined, 1
 - differential-equation models, 2–3
 - industrial applications, 1, 3
 - interactive modeling, 3–4
 - real world applications, 29–30
 - simple applications, 12–22
 - simulation software, 4
- Easy5™, 4
- End-of-run sample values, 90
- Engineering applications, 29–30, 42, 54–55
- Equation-oriented simulation programs, 4
- Error messages, 5, 12 n3, 63, 67–68, 70, 75, 77
- Euclidean norms, 67–68, 126, 128
- Euler integration rule, 10–11, 47, 112
- EUROSIM, 22, 171
- Event prediction, 45–46
- Expected values, 89, 93, 117
- Experiment protocol
 - commands, 89
 - default, 66
 - loop, 14
 - in neural networks, 128, 133, 166–167
 - primitive, 87
 - program, 3–4
 - scripts, *see* Experiment protocol scripts
 - signal generators/modulators, 57
- Experiment-protocol scripts, significance of,
 - 8–9, 11, 14–15, 20, 25, 28, 51, 60, 64, 67, 69–71, 77, 81–83, 87, 89–91, 100, 102, 182–184
- Extrapolation formula, 45
- Fast compilation, 5
- Fast Fourier transforms, 209
- Feedback, multilayer neural networks, 154
- Field-buildup delay, 22
- Finite-impulse response (FIR), 158–159
- First-order ordinary differential equations, 2
- Fixed-point block-diagram simulation
 - languages, 55
- Fixed-step integration, 8–9, 14, 47, 83–84
- Flat spots, 87–88
- Flight simulations, 74, 123
- Floating-point arithmetic, 5
- Floating-point operation, 49–51
- Fortran applications, 4, 63 n6, 71, 188, 194
- Fourier transform, 25. *See also* Fast Fourier transforms
- Frequency resolution, 57–58
- Frequency-response plots, 5
- FSCL (frequency-sensitive competitive learning)
 - algorithm, 148
- Full-wave rectifier, 49–50
- FUNCTION** declarations, 75–76
- Fuzzy-basis-function network, example of,
 - 203–205
- Fuzzy logic
 - models, 67, 71
 - rules tables, 172–174, 178–179, 183–184
 - servomechanism control example, 182–187
 - set techniques, 174–179
 - vector models, 179–182
- Gain, 22, 28, 38, 167
- Gambling returns study, 109–113
- Gamma delay line layer, neural networks, 157, 162–163
- Gaussian bumps, fuzzy set partitions, 176, 180
- Gaussian distribution, 92, 112, 145
- Gaussian noise, 89
- Gaussian probability density, 115
- Generalization, influential factors, 140
- Generators
 - fuzzy logic, 172–186
 - pseudo-random, 67, 90, 112
 - signal, 56–58
- Geographical information system, 197–199
- Global optimization, 86
- Gram-Schmidt algorithm, 132 n7
- Graphs, 85, 189
- Greville algorithm, 132 n7
- Grids, agroecological models, 197–199
- Guided-missile simulation, 25–29
- Half-wave rectifier, 42
- Hamming norms, 67–68
- Hardware in the loop, 2
- Harmonic oscillators, 64
- Heat conduction equation, 190, 192
- Heat exchanger model, 194–197
- Heuristic defuzzification assumptions, 178
- Heuristics
 - integration-step control, 46–47
 - pseudorandom-noise testing, 121
- Hewlett-Packard signal generator, 56–57
- Hydrodynamic-moment coefficients, 25
- Hysteresis models, 53–56
- if/then/else** statements, 20
- Image processing, 71
- Index-shifted
 - limiter function, 180

- neural vectors, 157–158, 168
- vectors, 63, 66–67
- Inertia, 25
- Initial conditions, 86, 88
- Initial value(s), significance of, 2–3, 5, 14, 26, 28, 34, 38, 53, 61, 71–72, 83, 90, 92, 107, 117
- In-line code, 77
- Input/output operations, 5, 9
- Integral absolute error (IAE), 24
- Integral squared error (ISE), 24
- Integrate-and-fire neural network, 164–166, 168
- Integrated squared error (ISE), 86–87, 90
- Integration
 - numerical, *see* Numerical integration
 - output-limited, 50
 - routines, 28, 36, 45–46, 57, 172
 - rules, 10–11, 83–84, 207–208
 - steps, 11–12, 45
- Interactive modeling, 3–4
- Inverse-square-law, 15
- Invocation arrays, 75–78
- Journal files, 85
- Kernel function, 98–99
- Landscape modeling, 71
- Law of large numbers, 88 n3
- Learning
 - biological, 125
 - competitive, 146–147, 150
 - momentum, 140
 - supervised, 154–155
- Levenberg-Marquart algorithm, 141
- Library functions, implications of, 42, 44, 50–51, 58, 61, 89
- Library submodels, fuzzy set partitions, 181
- lim**, 42, 43
- Limiters
 - characteristics of, 42–44
 - difference equations and, 49–50
 - DYNAMIC program segments, 78–79
 - fuzzy-set partitions, 180–181
 - outputs, numerical integration, 45–46
- Linear controllers/linear control systems, 22, 24–25, 74–75
- Linear filters, 158–159
- Linear harmonic oscillator, 12–15
- Linear transformations, 63, 72, 74
- Linux, applications of, xiii n1, 4, 6, 82, 113, 197
- LMS (least-mean-squares) algorithm, 132–133, 136, 139, 144, 159, 163
- Logarithmic scaling, 171–172
- Low-pass filter, 108
- LVQ (learning vector quantization) algorithm, 154–155
- Manufacturing-tolerance effects, 91–92
- Masking, of vector expressions, 69
- Mass-spring
 - dashpot system, 12
 - submodel, 78
 - systems, 64
- Mathematical analyses, 1
- Mathematical laws, 88 n3
- Mathematical models, 24
- MATRIX**, 69
- Matrix operations
 - characteristics of, 5, 69–71
 - difference equations in DYNAMIC program segments, 70–71
 - using equivalent vectors, 71
 - in experiment protocol scripts, 69–70
 - matrix assignments in DYNAMIC program segments, 70–71
- Matrix-vector products, 63–64
- Maximum/minimum
 - selection, 49–50, 68–69, 98
 - value-holding, 53
- M-dimensional combined-sample vector, 103
- Mean, 89, 99, 112
- Mean square
 - errors, 121–123, 140, 143
 - regression, 131, 138–139, 141–144
 - template-matching error, 146
- Measure statistics, *see* Statistics
- Membership functions, fuzzy set logic, 176–179, 181–187
- Memory networks, 67
- Method of lines (MOL), 63, 186, 188–191
- Microsoft Windows™, applications of, xiii n1, xiv, 7, 197, 199
- Minimum/maximum, fuzzy-set logic, 182
- Model replication, 4, 62–63, 87
- Modelica language, 4
- Modeling, interactive, 3–4. *See also specific types of models*
- Modulation, *see* Signal modulation
- Monte Carlo simulation
 - alternative to, 121–123
 - applications of, xiii n1, 1–2
 - characteristics of, 81
 - control-system errors caused by noise, 119
 - interactive, 96
 - optimization, 119, 121
 - repeated-run, 89–95, 97, 100, 102–103

- sequential studies, 91
- vectorized, 62, 93–97, 100–103, 112, 117–119
- Motor field delay, 22–24
- Multilayer neural networks
 - backpropagation, 138–141, 155
 - counterpropagation, 147, 155–156
 - neuron-model replications, 166–168
 - radial-basis-function networks, 141–146, 155
- Multiple parameters, 84–85
- Multiple simulation runs, splicing, 20–22
- Multirate sampling, 9
- Multistep integration rules, 10
- Nested submodels, 78
- Neural network(s)
 - characteristics of, 67, 106
 - competitive learning, supervised, 154–155
 - delay-line input layer, 157–163, 168
 - illustrations of, 126
 - linear, 127
 - with memory, 155–163
 - multilayer, 128–130, 138–146
 - nonlinear multilayer, 138–146, 159–161
 - pattern classification, *see* Pattern classification
 - in neural networks
 - pulsed-neuron replication, 153, 163–168
 - regression classification, 130–131
 - simulations, 63, 69, 71, 125–130
 - static, 157–163, 167–168
 - submodels, 141
 - training, 126, 134, 136–137, 140, 142–143, 155
 - vector models, *see* Neural network vector models
- Neural-network vector model
 - a posteriori* probabilities, 134–137
 - characteristics of, 125–127
 - exercising, 129–130
 - gamma delay line layer, 162–163
 - integrate-and-fire model, 164–166, 168
 - nonlinear, 138–141, 159
 - neuron-model replication, 166–167
 - pattern-matching-error, 136–137, 148–149
 - prototypes, 132, 138, 148–149, 152
 - quantization, 150–151
 - radial-basis-function networks, 141–146
 - simple operations, 126–127
 - successive sets, 130
 - tapped delay line, 157–159, 162
- Noise
 - characteristics of, 105
 - continuous, 107–109
 - control-system simulations, 116–119
 - input test, 116–118
 - Monte Carlo simulations, 109–116
 - pseudorandom, 10, 93, 109, 121
 - quantization, 51
 - random parameters and, 89
 - sampled-data random processes, 105–107
 - simulated, 109
 - vectorization and, 62
- Nonlinear controllers, 22
- 1776 cannonball, Monte Carlo simulation studies
 - with air turbulence, 113, 116
 - gun-elevation error effects, 91–95
- Nonlinear floating-point operation, 49–50
- Nonlinear oscillator, 15
- Normalization
 - fuzzy-set partitions, 175–180, 182
 - multilayer neural networks, 154
 - neuronal-layer patterns, 128
- Nuclear-reactor simulation, 72–74
- Null matrices, 69
- Numerical integration, 10–11
- OPEN DESIRE, overview of
 - CD contents, 210
 - command scripts, 5
 - command windows, 6–7
 - defined, xiii n1
 - Linux applications, xiv, 4, 6
- Open Source programs, xiii n1
- Operator integration-step control, 46–47
- Optimization
 - global, 86
 - implications of, 86–88
 - Monte Carlo, 119–121
 - neural-network patterns, 133, 141, 143
 - studies, 67, 69, 87–88
- Oscillators
 - harmonic, 64
 - linear harmonic, 12–15
 - resonating, 64
- Out-of-order assignment, 12
- Output overshoot, 22
- OUT** statements, 10, 12, 36, 46, 51, 65, 78–79, 108
- Oversimplified models, 30
- Overtraining, 140
- Overwriting, 191–192
- Parabolic linear partial differential equations, 192
- Parameter-influence studies
 - change effects, 80–81
 - characteristics of, 2, 62, 69
 - crossplotting, 86–87
 - model replication, 82–84, 87
 - multiple parameters, 84–85
 - optimization studies, 87–88, 119, 121

- programming, 85–88
 - random processes/parameters, 88–89
 - repeated-run studies, 81–82, 86
 - successive parameter settings, 85
 - system effectiveness, 85–86
- Parameter-sensitivity equations, 81
- Partial derivatives, 122–123, 188–191
- Partial differential equations
 - characteristics of, 63, 67
 - generalizations, 192–194
 - heat-conduction equation in cylindrical coordinates, 190, 192
 - hyperbolic, 197
 - linear, 192
 - method of lines (MOL), 186, 188–191
 - nonlinear, 192
 - simple heat-exchanger model, 194–197
- Parzen-window estimates, 99–101
- Pattern classification in neural networks
 - adaptive-resonance emulation, 151–153
 - associative memory and, 132, 134–138, 168
 - binary-selector, 138–139, 150–151, 155
 - characteristics of, 131
 - competitive, 146–154
 - conscience algorithms, 148
 - experiments with, 149–150
 - linear, 132
 - LMS algorithm, 132–133, 136, 139, 144, 159, 163
 - multilayer networks, 138–146
 - prototypes, 138, 148–149, 152
 - row matrices, 129–130
 - softmax image classifier, 133–137
 - supervised competitive learning, 154–155
 - template-pattern matching, 146–154
 - unsupervised, 147–149
- Pattern-row matrices, in neural networks, 129–130, 133–134, 145
- Performance measures, 90
- Perturbations, mean-square, 122–123
- Phase modulation, 58
- Phase-plane plot, 11, 14–16
- Physics applications, 54, 71–72
- Piecewise-continuous triangles, 180–181
- Piecewise-linear libraries, 42, 44
- Polynomials, 11
- Population-dynamics models, 18–20
- Prediction error, 161–162
- Predictor networks, 67
- Predictor variable, 34
- Preprocessors, block-diagram, 4
- Probability
 - applications, 96
 - density, 90, 97–100, 103, 112–113
 - distribution, 109
 - models, 88–89
 - theory, 1, 111
- Program-loop overhead, 96
- Prototype
 - failures, 30
 - vectors, 132, 158
- Pseudorandom noise
 - characteristics of, 35–36
 - generator, 67, 90, 112, 121
- Pulse width-modulated pulses, 58
- Radial-basis-function networks, 155, 201–203
- ran()**, 106–107, 109, 112, 116
- Random noise, 109
- Random parameters, 89
- Random processes, 88–89
- Random time-function inputs, 89
- Random variables, 103
- Random-walk simulation, 112–115
- Ranges, 96–97
- Real-world applications, 29–30
- Rectangle-window estimate, 98–99
- Rectangular matrices, 70
- Recursion, illegal, 66 n7, 158
- Recursive
 - assignments, 12 n3, 34, 51–52, 164
 - averaging, 107
 - function calls, 76
 - submodels, 78
- Reference manual, as information resource, 70
- Regenerative feedback, 55–56
- Regression
 - coefficients, 90, 97
 - neural networks, 131, 138–139, 141–142
- Relative frequency, statistical, 96–97
- Relay-comparator function, 44, 166
- repeat/until** statements, 20
- Repeated-run studies, 4, 81–82, 86
- Replicated-model studies, 87
- Replication, pulsed-neuron, 153, 163–168
- reset** statements, 3–4, 38, 52, 62, 71, 90
- Robotics, 3
- Root-locus plots, 5, 25
- Rotation matrices, vectors, 72–74
- Runge–Kutta integration rules, 10, 14, 45, 47, 84
- Runtime
 - compiler, 5
 - statistics, 28, 96, 113
- Sampled-data
 - assignments, *see* Sampled-data assignments
 - code, 35

- error measurement, 40
- operations, 10
- time average, 106–107
- Sampled-data assignments
 - applications, 32–33, 36, 83
 - continuous-variable differential equations with
 - switching and limiter operations, 51–58
 - difference equations and, 32–38
 - integration-step control, operator and heuristic, 46–47
 - user-defined functions, 78–79
 - vector, 64–66
- Sample-hold operation, 36
- SAMPLE m** statements, 9, 12, 36–40, 46, 51, 65, 78–79, 108
- Sampling
 - characteristics of, 5, 8–10
 - periodic, 107–108
 - points, 28
 - rate, 38
 - times and integration steps, 11–12, 36
- SAMT (Spatial Analysis and Modeling Tool)
 - database, 197–198
- Sat**, 42, 43
- SAT**, 42, 43
- Saturation
 - implications of, 22–24
 - plant simulation with digital PID controller, 40–41
 - unit-gain limiter, 42
- Sawtooth waveform, 54, 58
- Scalar assignments, 63, 66
- Scalar differential equations, xiii n1
- Scalar parameters, 83
- Scaled simulations, 2
- Schiesser differentiation operator, 189, 194
- Schmitt trigger, 47, 54–57
- Scicos, 4
- Second-order Runge–Kutta integration, 47
- seed m** command, 89
- Servo error, 47
- Servomechanism
 - bang-bang, 47–49
 - damping coefficient, 87
 - electrical, control system, 22–24
 - fuzzy-logic control of, 182–187
 - nonlinear, 116
 - vectorized Monte Carlo simulation, 120
- sgn**, 42, 44
- Shift registers, 67
- Short-term memory, 155, 157
- Signal(s)
 - generators, 56–58
 - modulation, 56–58
 - processors, 50–51
 - quantification, 50–51
- Simple applications, examples of
 - multiple simulation runs, splicing, 20–22
 - oscillators, 12–15
 - population dynamics models, 18–20
 - space-vehicle orbit simulation, 15–18
- Simple backlash models, 53–54
- Simulink™, 4
- Sinusoidal signals, 58, 108
- Smooth
 - approximations, 45 n7
 - curve, 98
 - fuzzy-basis functions, 181
- Smooth-interpolation assumptions, 45
- Software programs, simulation, 4. *See also specific software programs*
- Solaris®, xiii n1
- Solid-state ac motor controllers, 45
- Sort errors, 66
- Source code, xiii n1
- Space-delimited text files, 85
- Space-derivative submodels, 192, 194
- Space-vehicle
 - on-off vernier control rockets, 54
 - orbit simulation, 15, 17–18
- Square matrices, 69
- Square waves, 57, 79
- Squashing functions, 127
- STATE** declarations, 78
- State-equation models, 2, 5, 12, 74–75, 83
- State variable(s)
 - characteristics of, 3, 5, 11–12, 14, 28
 - difference-equation programs, 33–34, 36, 38
 - differential-equation, 62, 116
 - logarithmic scaling, 172
 - matrices, multilayer neural networks, 140–141
 - Monte Carlo simulation, 100
 - noise, 116
 - plant simulation with digital PID controller, 40
 - sampled-data assignments, 46, 52
 - values, 11, 80
- Statistical applications
 - measures, 67, 88
 - Monte Carlo simulation and, 89–97, 111
 - random-walk, 112
 - repeated Monte Carlo simulations, 89–90
- Steady-state condition, 15–16, 19
- Steering-moment coefficient, 25
- Step-function inputs, 11
- step** statements, 46–47, 52, 78–79
- Stiffness factor, 172

- Stiff systems, 191
- store/get** operations, 83–84
- Stripchart display, 23, 47
- sTrue** signal, 161–162
- SUBMODEL** declarations, 77–78
- Submodels
 - declaration and invocation, 76–78
 - with differential equations, 78
 - vectorization, 83
- Subpopulations, 18
- Subscripted variables, 66
- Subvectors, 71
- Successive approximation, 144
- Sums, 67
- Supervised learning, 126, 154–155
- Switches
 - characteristics of, 42, 44, 78–79
 - frequency resolution, 57
 - multiple, 46
 - outputs, numerical integration of, 45–46
- switch**, 42, 44
- Synapse, neural networks, 163–164

- Table construction, 2, 85
- Table-lookup functions, 58, 61
- Tangent functions, soft-limiting hyperbolic, 22
- Taxicab norms, 67–68, 126, 128
- Taylor-series expansion, 122
- Temperature, significance of, 86, 189, 192, 196
- Template matching, 146–154
- Test
 - signals, 57–58
 - statistics, 90, 97
 - testing programs, 89
- Tevent**, 45
- Threshold
 - level, 167
 - value, 163
- Time averaging, 106–107, 109
- Time delays, 67, 83
- Time histories, significance of, 5, 17, 24, 26, 39, 47, 74, 80–81, 85–86, 96–97, 113, 117, 143, 162, 165, 184
- Time relationships, differential-equation systems, 36
- Time-scaling operations, 172
- TMAX**, 2, 8–9, 11, 14, 16, 25, 28, 32, 38, 40, 47, 108, 112
- Torpedo, guided-missile simulation programs
 - complete, 28–29
 - with digital control, 38–41
 - multirun studies, 28
 - simple, 25–28
- Torque, 22–24, 40
- Track-hold operation, 52, 92–93
- Trajectory plots, 72
- Transmitters, neural networks, 163–164
- Triangle
 - functions, fuzzy-set partitions, 180–181, 184
 - wave, 57, 79
 - window estimate, 98–99
- TRIGA reactor, 74

- Undefined variables, 66
- Unit-gain saturation limiter, 42
- Unit matrices, 69
- Universal approximator, 140
- Unix, applications of, xiii n1, 4
- Updating assignment, 34
- User-defined functions, 58, 61, 75–76, 78–79

- Variable-order integration rule, 18
- Variable-step
 - implicit integration rule, 191
 - integration, 8–9, 11, 15, 17–18, 45, 47, 84
- Variance, 89, 93, 99, 112
- Vector**, 60
- Vector(s), *see specific types of vectors*
 - assignment, 68–69, 96
 - characteristics of, 5
 - differential equations, xiii n1
 - equivalent, 71
 - expression, 69
 - linear operations on, 72–73
 - models, multidimensional fuzzy-set partitions, 181–182
 - neural-network layers, 125–127
 - noisy-control-system simulation, 116–118
 - operations, *see* Vector/matrix operations/submodels
 - perturbation, 122
- Vector/matrix operations/submodels
 - differential equations, 59–66
 - Euclidean norms, 67–68
 - Hamming norms, 67–68
 - index-shifted vectors, 66–67
 - linear transformations, 72, 74
 - masking vector expressions, 69
 - maximum/minimum selection, 68–69
 - nuclear-reactor simulation, 72–74
 - rotation matrices, 72–73
 - sums and **DOT** products, 67
 - taxicab norms, 67–68
 - vector assignments, 59–66, 72, 74
- Vectorization, 4, 62–63, 81–83, 85, 87. *See also* Vectorized simulation

- Vectorized simulation
 - agroecological models, 197–199
 - EUROSIM, 171
 - fuzzy-logic function generators, 172–186
 - with logarithmic plots, 171–172
 - partial differential equations, 186–197
- Vectr d/dt**, 61
- Velocity, implications of, 20, 25, 59, 91–92
- VissimTM, 4
- Voltage, 22, 47, 120, 163–164
- Volterra-Lotka differential equations, 19–20
- Voronoi tessellations, 150–151
- Wind-tunnel data, 122–123
- Yaw-rotation equation, 25