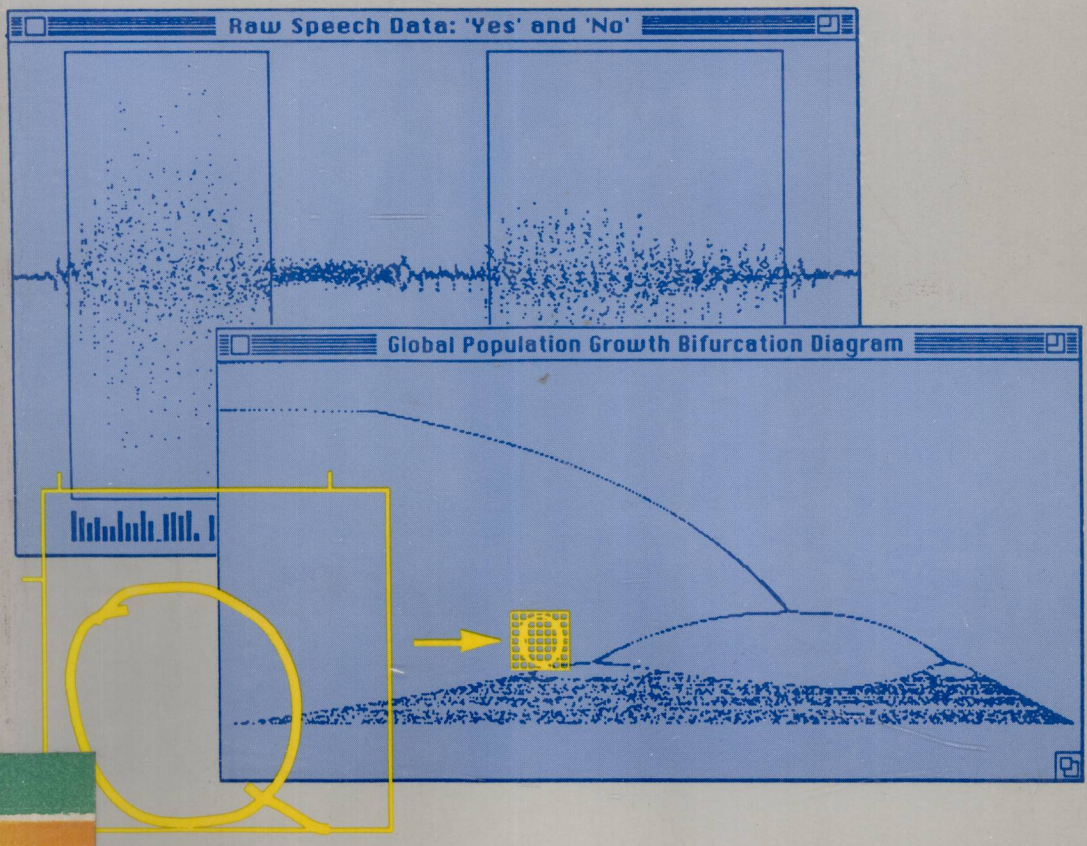
Mark Watson

Common LISP Modules

Artificial Intelligence in the Era of Neural Networks and Chaos Theory



Springer-Verlag

Mark Watson

Common LISP Modules

Artificial Intelligence in the Era of Neural Networks and Chaos Theory

With 35 Illustrations

Springer-Verlag
New York Berlin Heidelberg London
Paris Tokyo Hong Kong Barcelona

Mark Watson
Science Applications International Corporation
10260 Campus Point Drive
San Diego, CA 92121
U.S.A.

Printed on acid-free paper.

Copy produced using the author's PostScript file.
Printed and bound by Book-mart Press, North Bergen, New Jersey.
Printed in the United States of America.

9 8 7 6 5 4 3 2 1

ISBN 0-387-97614-0 Springer-Verlag New York Berlin Heidelberg
ISBN 3-540-97614-0 Springer-Verlag Berlin Heidelberg New York

Preface

While creativity plays an important role in the advancement of computer science, great ideas are built on a foundation of practical experience and knowledge. This book presents programming techniques which will be useful in both AI projects and more conventional software engineering endeavors. My primary goal is to entertain, to introduce new technologies and to provide reusable software modules for the computer programmer who enjoys using programs as models for solutions to hard and interesting problems. If this book succeeds in entertaining, then it will certainly also educate.

I selected the example application areas covered here for their difficulty and have provided both program examples for specific applications and (I hope) the methodology and spirit required to master problems for which there is no obvious solution.

I developed the example programs on a Macintosh™ using the Macintosh Common LISP™ development system capturing screen images while the example programs were executing. To ensure portability to all Common LISP environments, I have provided a portable graphics library in Chapter 2.

I will provide Floppy disks (Macintosh™ or IBM PC™ format) containing the source code to all example programs listed in this book for $12 (order from my address below). This software contained in this book is "as is" with no warranties or proof of correctness; the user of this software accepts all liabilities for its use. I also welcome other correspondence (preferably via electronic mail). I can be contacted on the following public information services:

Compuserve address: 75765,556

BIX: wmark

Home address: 535 Mar Vista Dr. Solana Beach, CA 92075

Acknowledgments

I would like to thank my wife Carol for encouragement while preparing this book; my company, SAIC, for supporting my research in AI and Neural Network technologies; Tim Kraft, George Works, and David Rumelhart for many discussions concerning neural network technology; Bob Beyster, Carl Rindfleisch, John Benepe,

John Thompson, Bobby Hunt, John Penhune, Tom Bache, Duane Knize, Larry Kull, Larry Hunt, and Jim Martin for supporting my AI research at SAIC; Carol Watson, Cris Kobryn, and Tim Kraft for reviewing rough drafts of this book; my father, Ken Watson, for the use of his laser printer in preparing this book; my agent, Bill Gladstone, and his assistant, Matt Wagner; my editors Gerhard Rossbach, Suzanne Anthony, and Kenneth Dreyhaupt; and Nancy Wilson for the excellent job editing my book. I would also like to thank the authors of all the books in the bibliography section. I have learned a great deal from them.

Contents

1
Introduction

Computer programmers have tried to simulate intelligent behavior with programs since the invention of computers. It is easy to be sympathetic with pioneering computer scientists who predicted human-class cognitive performance out of computers when their first experiences showed that even early computers' arithmetic ability far surpassed that of humans. With the benefit of hindsight, it is clear that artificial intelligence (AI) practitioners promised too much too soon.

We now know that human beings far outclass even supercomputers in pattern recognition and associative memory recall. A common (and much enjoyed) debate deals with the likelihood of AI hardware/software systems rivaling humans in cognitive tasks. This debate will not be covered in this book! The reader of this *book* will gain insight into the creative process needed to break down old technology barriers and will develop an optimistic "can do" attitude.

1.1 Overview

Except where noted, the chapters in this book can be referenced independently. Chapter 2 contains the device independent graphics primitives used in this book. The neural network simulator developed in chapter 3 forms a conceptual basis for all subsequent treatment of neural networks in chapters 4 through 7. The material in chapter 9 is based on the natural language parsing techniques presented in chapter 8.

Each chapter is generally structured as: background information, theory, a discussion of sample programs, program listings, program output, additional information on the execution of the sample programs, and suggested projects. There are five sections in this book:

1.1.1 Neural networks

Chapter 3 provides background to the theory and usefulness of neural networks, and presents a neural network simulator which will be used in a speech recognition system in chapter 5 and a handwriting recognition system in Chapter 6. Chapter 4 introduces associative memory systems using Hopfield neural networks. Chapter 7 introduces the biologically inspired neural network model ART2 of Stephen Grossberg and Gail Carpenter.

1.1.2 Natural language processing

Chapter 8 introduces syntax-based parsing techniques with an animated parsing program. Chapter 9 adds semantic processing to syntactic analysis and compares and contrasts the example library database query program to programs using Conceptual Dependency theory.

1.1.3 Expert-systems

Expert-systems, usually embedded in larger software systems, are becoming a standard software engineering tool. Sample programs in chapter 10 demonstrate forward and backward chaining techniques. Advice is offered in the evaluation of commercial expert-system shells.

1.1.4 Search

A software module for performing A* heuristic network searches is developed in chapter 11. A simple but effective chess playing program is developed in chapter 12.

1.1.5 Chaos theory

Chapter 13 introduces chaos theory. Chapter 14 contains sample programs for generating Mandelbrot set plots and iterated function system plots.

1.2 Prerequisites

Ideally this book should be read with the benefit of a computer system with a LISP compiler since all chapters contain, as program miniatures, small programs written to illustrate an idea or concept. Note: All programs are implemented in Common LISP. However, the examples should be easily portable to other LISP dialects. The neural network, and fractal examples can be ported to other languages like C and Pascal since they do not rely on the symbolic features of the LISP programming language.

This book assumes a reading knowledge of LISP and the implementation of a set of graphics primitives described in chapter 2 for creating a graphics window and performing simple graphics operations in this window. This set of primitives can be written in about two pages of code in any Common LISP system that provides simple graphics support; once this library is written, all of the miniatures presented

in this book should execute in all Common LISP environments using a LISP listener window for user interaction with graphics going to a separate window.

1.3 Conventions

Key words appear in **bold** when first introduced. All program literals will be in *italic* when they appear in text and in comments in the program listings. All program test outputs have user responses in **bold** text with program output shown in normal typeface. Long program comments will be blocked off separately and indented to the same column as the surrounding text. Short comments appear on the same line as LISP code. Example programs are printed in Helvetica type.

2
Basic Software Tools: Machine-Independent Graphics

One of the best uses for a high-level language like Common LISP is in the rapid prototyping of window-based user interfaces, now an accepted part of the software development process. Common LISP is especially appropriate for developing user interfaces when augmented with one of the current object oriented programming (OOP) packages: Object Lisp, Common LISP object system (CLOS), or Portable Common Loops (PCL).

However, one requirement for a programming language book is to enable readers to effortlessly get the sample programs in a book up and running (with no modifications!) on their computer and within their language environment. For this reason, the example programs in this book will use a simple set of graphic primitives and will not use any OOP language extensions despite their obvious benefit. Once the graphics library described below is implemented, all example programs should execute without modification on any system.

The examples in this book use the following graphics primitives:

init-plot – Creates a graphics window
plot-fill-rect – Fills a rectangle with a gray-scale value
plot-size-rect – Plots a rectangle whose size is specified by a gray-scale value
clear-plot – Clears the graphics window
pen-width – Sets the default line-plotting width
plot-frame-rect – Draws a rectangle
plot-line – Draws a line
show-plot – Makes visible and unobscured the graphics window
plot-string – Plots a string of text
plot-string-bold – Plots a string of bold text
plot-string-italic – Plots a string of italic text
plot-mouse-down – If the mouse is clicked, returns the x-y click position

Here is an example implementation of these primitives for the Macintosh running Macintosh Common LISP:

```
;;
; Common plot routines for Common LISP Compatibility — for MACINTOSH
;
; Externally callable functions:
;
; init-plot( )                              ;; creates a graphics window
; plot-fill-rect(x y xsize ysize value)     ;; fills a rectangle with a gray-scale value
; plot-size-rect(x y xsize ysize value)     ;; plots a rectangle value pixels wide
; clear-plot( )                             ;; clears the graphics window
; pen-width (nibs)                          ;; sets the pen drawing width
; plot-frame-rect(x y xsize ysize)          ;; plots a frame rectangle
; plot-line(x1 y1 x2 y2)                    ;; plots a line between two points
; show-plot( )                              ;; shows graphics window
; plot-string(x y str)                      ;; plots a string at position (x y)
; plot-string-bold(x y str)                 ;; plots a bold string at position (x y)
; plot-string-italic(x y str)               ;; plots a italic string at position (x y)
; plot-mouse-down( )                        ;; returns position of mouse click
;;

;;
; Initializes a standard plot window:
;;

(defun init-plot (&optional (title "Plot Window") (xSize 250) (ySize 250) )
  (setq *w*
        (oneof *window* :window-title title
                        :window-size (make-point xSize ySize)
                        :window-type :document-with-zoom)))

;;
; Fills in a rectangle with one of five gray-scale values:
;;

(defun plot-fill-rect (x y xsize ysize pattern)
  (setq pattern (truncate pattern))
  (let ((ppp *black-pattern*))
    (if (< pattern 1)
      (setq ppp *white-pattern*)
      (if (equal pattern 1)
        (setq ppp *light-gray-pattern*)
        (if (equal pattern 2)
          (setq ppp *gray-pattern*)
          (if (equal pattern 3)
            (setq ppp *dark-gray-pattern*)
            (setq ppp *black-pattern*)))))
```

```
    (ask *w* (fill-rect ppp x y (+ x xsize) (+ y ysize)))))

;;
 ; Makes a black rectangle of size proportional to val. This is an alternative
 ; to using function plot-fill-rect for showing graphically the value of a number.
 ;;

(defun plot-size-rect (x y xsize ysize val)
  (setq val (min val xsize))
  (ask *w* (erase-rect x y (+ x xsize) (+ y ysize)))
  (ask *w* (fill-rect *black-pattern* x y (+ x val) (+ y val))))

;;
 ; Clears (erases) the plot window:
 ;;

(defun clear-plot ()
  (ask *w* (fill-rect *white-pattern* 0 0 400 400)))

;;
 ; Sets the drawing size for the pen:
 ;;

(defun pen-width (nibs)
  (ask *w* (set-pen-size (make-point nibs nibs))))

;;
 ; Frames a rectangle of size (xsize ysize) at position (x y):
 ;;

(defun plot-frame-rect (x y xsize ysize)
  (let ((x2 (+ x xsize))
        (y2 (+ y ysize)))
    (ask *w* (frame-rect x y x2 y2))))

;;
 ; Draws a line between two points:
 ;;

(defun plot-line (x1 y1 x2 y2)
  (ask *w* (move-to x1 y1))
  (ask *w* (line-to x2 y2)))
```

```
;;
 ; Shows plot window if it is obscured:
 ;;

(defun show-plot ()
  (ask *w* (window-select)))

;;
 ; Plots a string at position (x y):
 ;;

(defun plot-string (x y str &optional (size 10))
  (setq x (truncate x)  y (truncate y))
  (ask *w* (set-window-font (list "Times" :plain size)))
  (ask *w* (move-to x y))
  (princ str *w*))

;;
 ; Plots a string in bold font at position (x y):
 ;;

(defun plot-string-bold (x y str &optional (size 12))
  (setq x (truncate x)  y (truncate y))
  (ask *w* (set-window-font (list "Times" :plain :bold size)))
  (ask *w* (move-to x y))
  (princ str *w*))

;;
 ; Plots a string in italic font at position (x y):
 ;;

(defun plot-string-italic (x y str)
  (setq x (truncate x)  y (truncate y))
  (ask *w* (set-window-font '("Times" :plain :italic 12)))
  (ask *w* (move-to x y))
  (princ str *w*))

;;
 ; Tests for a mouse down event (returns nil if the mouse button is
 ; not being held down when this function is called;  returns a list
 ; of the x and y screen coordinates relative to the plot window
 ; origin if the mouse button is being held down while this
 ; function is being called):
 ;;
```

```
(defun plot-mouse-down ()
  (mouse-down-p))

;;
 ; A simple test program:
 ;;

(defun test ()
  (show-plot)
  (clear-plot)
  (dotimes (i 6)
    (plot-fill-rect
     (* i 9)
     (* i 9)
     8 8
     i)
    (plot-frame-rect (* i 9) (* i 9) 8 8))
  (dotimes (i 50)
    (plot-size-rect
     (+ 160 (random 200)) (random 100) (random 20) (random 20) (random 5)))
  (dotimes (i 4)
    (plot-string (* i 10) (+ 150 (* i 22)) "Mark's plot utilities..."))
  (plot-string-bold 20 260 "This is a test... of BOLD")
  (plot-string-italic 20 280 "This is a test... of ITALIC"))
```

Before you implement these primitives within your Common LISP environment, note the following:

1. If you are not using Macintosh Common LISP, it will be necessary to recode each of the above functions using graphics functions provided by your Common LISP vendor. You can start by implementing *init-plot*, *plot-line*, and *plot-string*. The remaining functions can be "stubbed" and implemented later as you need them.

2. If the graphics library provided with your Common LISP environment does not support different fonts, use the function *plot-string* for drawing characters on the screen.

The screen image in Figure 2.1 shows the results of executing the function *test* at the end of the preceeding listing.

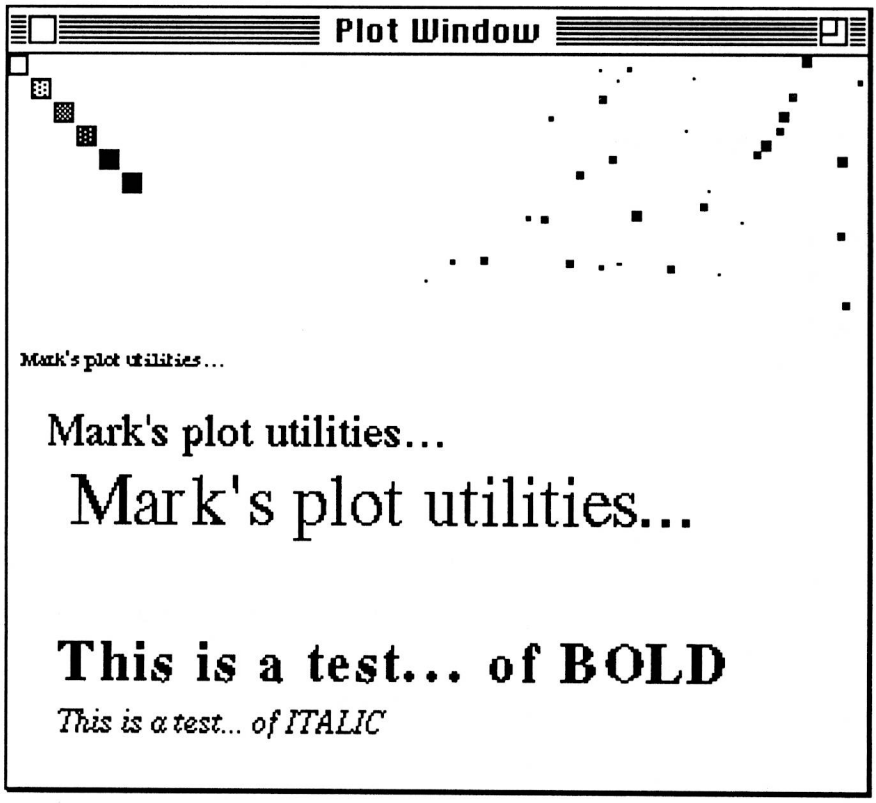


Figure 2.1. Sample screen image showing the execution of function *test* in the graphics primitives plot library.

3
The Substrates of Intelligence, a Neural Network Primer

What is an artificial neural network? How do artificial neural networks compare with conventional computers and traditional massively parallel computers, and when are they more useful? What are possible applications of neural network technology? These questions will be answered in the beginning of this chapter, followed by the presentation of an engineering model based on equations which characterize the behavior of one popular class of neural networks for **supervised** learning. Supervised learning uses both input training patterns and desired system output patterns for neural network training. This chapter then provides a simple program demonstrating how to write and run artificial neural network simulators followed by a listing of a complete production-capable neural network simulator. Examples show how to set up training data for and run this complete simulator (which is used for speech recognition in chapter 5 and reading handwritten characters in chapter 6). This chapter ends with more suggested projects and hints for their solution.

3.1 Background

Neural networks are systems of very simple processing elements that are massively interconnected. Long term memory or information content in neural networks is typically stored in the state of the interconnections, not the processing elements themselves. Currently artificial neural systems are computer simulations of systems containing large numbers of massively interconnected processors. Neural networks are not programmed in the traditional sense; they are **trained** by exposure to data values specifying desired system outputs for given system inputs.

Many AI problems can be solved in principle with known algorithms, but not in practice because of the limited processing capabilities of the Von Neuman computer model (as implemented in single processor IBM PCs, Apple Macintoshs, DEC VAXs, CRAYs, etc.). These conventional computer systems are best used for applications which are procedurally oriented like business and nonparallel scientific calculations. By contrast, neural systems exploit the inherent parallelism in many pattern-matching and recognition tasks. In the future they will be used to augment conventional computers, greatly speeding up pattern matching and cognitive tasks.

Neural systems are very fault tolerant; they can be partially destroyed and still function with some degradation of performance. This fault tolerance will someday