David W. Russell

The BOXES Methodology

Black Box Dynamic Control

Springer

The BOXES Methodology

David W. Russell

The BOXES Methodology

Black Box Dynamic Control

Springer

David W. Russell
The School of Graduate Professional Studies
Penn State University
Penn State Great Valley
30 East Swedesford Road
Malvern PA 19355
USA

ISBN 978-1-84996-527-9 e-ISBN 978-1-84996-528-6
DOI 10.1007/978-1-84996-528-6
Springer London Heidelberg New York Dordrecht

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2012931406

Cover design: eStudio Calamar S.L.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To Donna
Without whose encouragement this work
would never have been completed

Donald Michie: A Personal Appreciation

I met Donald Michie in Edinburgh in the early 1970s after reading some of his papers and became fascinated by a learning methodology that he called "The BOXES Method" and I continued to enjoy a fairly loose but valued relationship with him over the many years since; meeting at conferences, lunch in London on occasion, giving a lecture at the Turing Institute in Glasgow and so on. I have published 32 technical papers based on the BOXES method. Donald was a controversial, outspoken figure in UK academia over the years and seemingly always ready to branch out into new adventures. Among many others, I was much saddened by his much too soon departure from this life in July 2007 following an automobile accident. It was truly gratifying to see the appearance of a book titled *Donald Michie on Machine Intelligence, Biology and more* that contains a wonderful collection of Professor Michie's papers, and I quote from page 6:

> With the 1990's also came, finally, recognition of his contributions to machine intelligence in the form of several awards. In 1995, he co-founded the Human Computer Learning Foundation, with the aim of exploring ways in which computers could be used to assist and improve human skills [1].

Mechatronic machines do very well what we humans are extremely limited in doing; any task that involves flawless and reliable memory, undivided attention, and strict devotion to the task at hand. Donald opined that "… *The black death of our times is the world's escalating complexity*" and perhaps it is the profound simplicity of the BOXES method, and Donald's enthusiasm for everything, that was my motivation for writing this monograph.

Reference

1. Srinivasan, A (ed) (2009) *Donald Michie on Machine Intelligence, Biology and more*. By permission of Oxford University Press

Foreword

It is interesting how casual browsing through a technical article can capture one's imagination and alter a career. In 1973, while looking for reference materials to supplement the author's doctoral studies in real-time adaptive, intelligent control, his attention came upon a series of papers describing a new genre of research namely *learning machines*.

The author's own doctoral studies [1] had focused on the design of a real-time pattern generating automaton for the control of an experimental fast reactor, in which control was maintained by filling and emptying tubes of moderator fluid based on what, today, would be called rule-based control laws. Yet, it was also a sly game! The selection of not just *how many* control tubes needed to be activated to respond to a power demand, but it was also necessary that an even *distribution* of tubes be maintained for stability reasons, and that the choice of tube be made with minimum *disturbance*. Of course, in the fast reactor setting, the proximity of any tube to any other enhances the control strength of both. But enough about fast reactors—what was fascinating was that when it was necessary to make changes to the control setting, the pattern needed to change but with minimum alteration of tube activations. It was a sort of Connect Four$^{\text{TM}}$ situation in which one player's own move not only affects his/her board plan but potentially reverses and obliterates those of the opponent.

As in a board game, dynamic systems create control situations that can be calm, erratic, or even chaotic, and change without much, if any, warning. To keep up with these demands it became obvious that computational assistance would be necessary. This would need to have the capacity to perform not only the usual control functions but also have the *intellect* to solve logical problems. Around that time, machine learning was emerging, pioneered by Professor Donald Michie at Edinburgh and others. In this field of study, it was clear that the digital computer, with its speed of calculation could compare many options and return a good control decision in almost real-time.

But, was it doing more than running sophisticated trial and error sequences, keeping a tabulation of what was the *best so far*, until it ran out of time when it returned an *intelligent guess*? Artificial intelligence was on the horizon!

With faster hardware came a new slew of algorithms that seemed to be able to present fast solutions to puzzles, games, and conundrums that elude all but the smallest percentage of humankind. For example, as far back as 1959, Samuel [2] had developed a program that purported to have learned to play checkers and improved with each game. What was especially intriguing was the fact that the automaton (which was the computer program and an avatar to assert board entries and read opponents moves) improved in the performance of certain tasks (i.e. not losing or winning) over time. Around that era many software developers (too numerous to mention here) began to work on the resolution of various game situations in the much more complex game of chess which culminated in the now historical match [3] in 1997 between Deep Blue^{TM} and the then reigning World Chess Champion, Garry Kasparov. In 2010, the state of the art in chess playing software systems had reached such a level of proficiency that Veselin Toparov trained for the World Chess Championship using the Blue Gene [4] Supercomputer, although eventually losing to Viswanathan Anand in games that were all close.

What has fascinated scientists, especially engineers, is not only the answer to *is the machine truly intelligent* but also *can an automaton perform real-world tasks better than a human or analog system*?

Concurrent with this activity by what was to become the *Artificial Intelligence* community was a surge in interest in the replacement of analog process systems by what became known as adaptive, digital controllers. It was only a matter of time before the two fields overlapped and the era of learning systems began. For example, among many others, a 1968 paper by Lambert and Levine [5] appeared titled "Learning Control Heuristics." By replacing analog control systems by digital data collection systems it became obvious that the digital system could do much more than produce pretty graphs and digital meter readings. The era of analog controllers was coming to an end and the new world of control algorithms such as statistical trending, optimization, state space, and fuzzy logic had arrived. All the system needed was the ability of a human to program its rule sets, be it the exclusion rule for noughts and crosses (a.k.a. tic-tac-toe in the US) in which it is not legal (or nice!) to overwrite a square already held by the opponent, or the control limits on the temperature of a chemical process. A fatal flaw had emerged: Was all the digital processor doing replacing functions that an analog computer or human could do with some rather expensive electronic apprentice. Most systems required a mathematical model of the process under control that was derived from physical laws or known rules. The programmer's task was to code the algorithms and attach them to a waiting program. Did this introduce intelligence or just mimic the workings of analog systems? Was direct digital control (DDC) the answer to solving complex control problems that were rapidly exhausting the capacity of analog controllers?

The assertion that machines (vis-à-vis) robots possess intelligence was and is fiercely debated by engineers and philosophical purists alike. The now infamous Lighthill Controversy debate that was aired live by BBC TV in 1973 was in essence an 81 min argument between Donald Michie and leading researchers that is still blamed for the AI Winter, so called because of the bleak times AI researchers went through in terms of getting grants and support.

Over 30 years later, a 2007 posting in the Steeb-Greebling Diary [6] sadly reported the ongoing disagreement between Michie and his colleagues. As an outcome of that same debate, it recounts that Sir James Lighthill subsequently published a report that called a halt to artificial intelligence research in all but two areas. The resulting dissolution of Donald's research group in Edinburgh left him isolated in the research unit.

Michie left Edinburgh in 1984 to become Professor of Computer Science at Strathclyde University where he established the Turing Institute. He worked there for 10 years contributing much to the field of machine learning. In March 1990, the author visited Michie at the Turing Institute and gave a presentation based on the Liverpool work aptly titled *The Trolley and Pole Revisited.*

Machine intelligence is still a hot topic of conversation. As recently as 2008, the Guardian [7] ran a headline: "Artificial intelligence: God help us if machines ever think like people" in which Charles Arthur cites a paper by Gary Marcus [8] in which he states that there are two systems operating in our minds, one *ancestral* and the other *deliberative*. The deliberative system is younger genetically speaking but the older one, being better wired into our subconscious, often gets the upper hand. What this suggests is that human minds are not really able to create much that is new regardless of the challenge, so why should an artificial system that may be relentlessly pursuing some algorithmic agenda produce innovative, yet viable and practical solutions to a problem?

Returning to history around same this same time, a seminal work by Michie and Chambers [9] written in 1968 attracted the author's special attention. The latent thesis of the work was that *machines can learn* and researchers have been pursuing this and the obvious extension: *do machines think* to this day. The outcome of the revelation was the foundation of the research team at the then Liverpool Polytechnic in the UK and where this journey began.

This monograph is a collection of the author's personal research work and is most certainly not all inclusive. Some software design guidelines are included in Appendix B to encourage future work by others.

References

1. Russell, DW (1970) Advanced Analysis of Fluid Control Systems for Nuclear Reactors PhD thesis. Council for National Academic Awards, London
2. Samuel A.L. (1959) Some studies in Machine Learning using the game of checkers. *IBM J. of Res. Dev* 3, 210-229
3. Deep Blue: (1997) Game 6: May 11 www.research.ibm.com/deepblue/watch/html/c.shtml. (Accessed December 21, 2010)
4. Blue Gene. (2005) *IBM J. of Res. Dev.*: 49, No: 2/3 191-489

5. Lambert, J.D. and Levine, M.D.(1968) Learning Control Heuristics. *IEEE Trans on Automatic Control*: VolAC-13, 741-742

6. The Lighthill Controversy The Streeb-Greebling Diaries (2007) http://streebgreebling.blogspot.com/2007/07/lighthill-controversy.html. (Accessed November 20 2010)

7. Arthur, C. (2008) Artificial intelligence: God help us if machines ever think like people http://www.guardian.co.uk/technology/2008/jun/20/artificial.intelligence

8. Marcus, G. Kluge: (2008) *The Haphazard Evolution of the Human Mind*. Houghton, Mifflin Harcourt Publishing Company, New York: NY ISBN 978-0-618-87964-9

9. Michie, D. (1986) *On Machine Intelligence*. Ellis Horwood Ltd. Chichester, England. ISBN: 0-7458-0084-X

Acknowledgments

Special acknowledgement is due to Steve Rees and John Boyes, my intrepid Liverpool team members, through whose endeavor the BOXES automaton was designed, constructed and successfully operated. I recall the countless hours over four years that Steve tried to outplay the automaton manually and his glee at obtaining a controlled run of 10 s one day, when of course the automaton attained 10 s after an hour or so of training. I am also grateful to Dennis Wadsworth for his enthusiasm and skill in adapting the BOXES algorithm into a form that positively enhances database access and his contribution to Chap. 10.

I especially acknowledge the work of many fellow researchers who have used, altered, and otherwise adapted the BOXES method over the years and whose work may have been inadvertently omitted.

Lastly, I am indebted to my Springer editors and friends Anthony Doyle, Claire Protherough, Christine Velarde, and Grace Quinn, without whose help this book would never have come to print. All and any mistakes are mine alone and readers should please feel free to contact me with any suggestions you may have for improvements or correctness.

David W. Russell
Malvern, PA
drussell@psu.edu

Contents

Chapter 2
The Game Metaphor

2.1 Computers can be Programed to Play Games

Michie [1] points out that solving a puzzle (what he called a one player game) pits the wits of a single player against the rules of the apparatus upon which the puzzle is being staged be it physical or logical. Games on the other hand are usually played against at least one other player, even if that player is only nature itself. Puzzles can be deterministically logical, such as in Sudoko, or more spatial as in the logical rearrangement of rings in the Tower of Hannoi problem. How they are solved has contributed much to the foundations of learning theory. The heuristics of how a human or computer system customarily learns to play a game relies on an examination and understanding of the state of the game before and after each move regardless of who made it. In the preceding chapter some time was spent illustrating this by describing how the rules of chess, a fairly sophisticated but deterministic game, were programed into a digital computer with good result.

Provided the topography and status of the board is correctly described, one method that a program can determine how to play is to search through seemingly countless moves and assess the consequences of each possible variation before submitting a final selection. A more conservative player might play to not lose, while a nervous player might try to finagle a draw by forcing the board into the situation where every subsequent move only plays into a series of non-winnable states out of which neither player can escape and subsequently force the draw.

Provided the moves are accurately transcribed and that the players do not cheat, a human can play the program, or one program can play another program. Both players, with their varying skill levels, must scan the same board and determine appropriate legal next moves. Depending upon the game, moves can cause pieces to traverse many squares and be allowed even to skip between the rows and columns of the board.

The object of most games is really very simple … win. A novice player will fall for gambits, in which shameless entrapment masquerades as an error by the opponent. If the bait is taken, to the dismay of the minnow a drastic counter-move is made by the predator. While not always ending the game, severe damage to a defense or plan-of-attack usually ensues. Readers may be familiar with the fools mate in which seemingly innocuous piece positioning suddenly converges to checkmate after only three moves or so. By playing many chess games, normally over many years, it is possible for a human (or computer program) to become very proficient in taking on all comers and playing at a high level.

2.1.1 Playing by Rules

In the artificial intelligence (AI) field it is commonplace to write software that invokes game strategies but only to the depth of the analyst's knowledge of the game or the opponent. Rule-based systems (RBS) seek to mimic human expertise and generally contain a knowledge base [2], an inference engine and an interpretive language. The software must analyze the board position and having recognized it, point to and obey a rule set that has been previously associated with the current situation.

For example in a noughts and crosses game, if the paying arena is simulated with an array *Board{1..9}*, a move is denoted as *k*, and each player is assigned a *MyToken* value (usually "X" and "O"}, two simple rules might be:

```
RULE 1: Winning {k, MyToken}
   IF    WIN possible by placing {MyToken} in position k
   THEN          Board(k) = MyToken: Win = true
   ELSE          Win = false
   EXIT RULE

RULE 2: Avoiding a loss {k, MyToken}
   IF     Opponent WIN possible by placing NOT (MyToken) in position k
   THEN         Board(k) = MyToken
   ELSE         Select another move
EXIT RULE
```

The syntax in WIN would be implemented by a subprogram performing a scan of the potential win status on the board with the suggested move (k) temporarily inserted. Of course there may be many rules that fire for any board combination in more complicated situations or games. Much use is made of this AI method in automated medical diagnostic systems and many of the expert systems that are commercially available. By entering known clinical observations and patient symptoms a diagnostic engine can produce a list of possible illnesses which the clinician can use to decide upon the next treatment. The list may include false positives which the physician can eliminate. There are other data derived rule-based systems in which the results of many recursions of the game generate facts or options.

Rule-based systems are present in everyday systems. In an automobile speed control system, for example, if the driver requests a speed increment that is greater than some preset value, say 10 mph, the engine gear will usually be reduced by one, but only if the current gear is greater than second or less than seventh. The following fictitious algorithm could have been derived by automotive engineers as being the best for smooth pickup and avoidance of fuel wastage in high speed travel in the lower region of the drive train. If the vehicle is in a low or top gear, the request for rapid acceleration will be ignored and the vehicle motion proceeds in the same gear. The following is a primitive for such a rule that attempts to processes the directive <reduce gear> and smoothly obeys.

```
RULE: Speed control gear shift reduction
      IF    speed request > {10} miles per hour
                AND     current gear > 2 and current gear < 7
      THEN    < reduce gear > by 1
      ELSE   < ignore > shift down
EXIT RULE
```

While interesting, the book will focus on systems that in general either have too many rules to encode this way, or are dynamically ill-defined.

2.2 Reactionary Strategy Games

Games such as chess that are highly regarded as requiring much intelligence to play well, all have a common theme that the opponent must play in the same domain so that counter-moves can be formulated to thwart the opponent's plans. It is probably true that a *good* game requires two players of comparable skill. It is a matter of conjecture that a chess program could probably not reach grand master level if its only competition were novices. In a manner reminiscent of how expert systems are produced, it is the ability of the trainer that really defines the quality of the product. If a supercomputer is searching through all options in the search tree on every move, the so-called brute force approach, then it is its *speed* that makes it a formidable opponent rather than its cunning or skill.

The fault in systems that need feedback from an opponent in order to learn is that a devious player can mislead the automaton and in so doing confuse the logic that is designed to produce its skill. As the familiar Byzantine Generals problem [3] demonstrates, it is difficult to create a successful plan when one or more of the players may be *traitors* who violate the rules of interactive consistency. To illustrate this further, the following section describes a noughts and crosses algorithm and how a simple variation on the game would confound that noughts and crosses engine. Both games use an identical state space (game board) and vary only in the definition of winning.

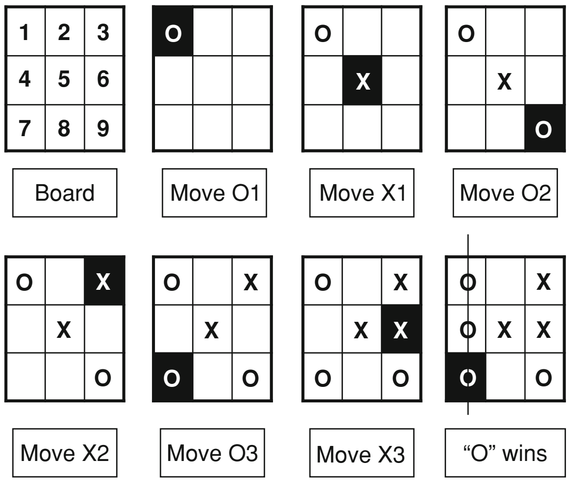


Fig. 2.1 A simple noughts and crosses game

2.2.1 Noughts and Crosses

In the noughts and crosses game it is the rule that one of the two players each of whom is assigned a different token (viz. "O" or "X") must get the same three tokens in a row, column, or diagonal set to win. Figure 2.1 shows a simulated game in which "O" wins rather easily partially due to poor play by "X".

It is apparent that once the "**X**" player makes "**MOVE X2**"—the "O" player will win regardless of what the other player does provided no further errors are made. In order to encode this game so that a computer can play, the board can be simulated as a (3 × 3) or better yet a (9 × 1) array as shown in the first square in Fig. 2.1. The logic of a simple noughts and crosses program is shown in Fig. 2.2 and each function block is expanded upon in Table 2.3 later.

The essential components of the simulator must have the ability to allow two opponents to play each other. A human player must *choose* the number of the board position {1 through 9} that is desired to be played next. The program checks that this position is blank to avoid cheating, if so the move is inserted, the number of available board positions is decremented by one, and a check made for an end-game situation. If the game is not over, the other user (the program) takes over as the player and the game repeats. The mechanism for *selecting* the next move by the computer is what is interesting and will reappear later in the BOXES algorithm. When it is the computer's turn, the program forms a list of all available board moves and then *selects* one of those. As shown later in Sect. 2.5 one method is to reference a payoff or value table and select the position with the highest score.

Using the game shown in Fig. 2.1 with the automaton (badly) playing "X" after "O" makes the first move, from the 8 moves left {i.e. 2, 3, 4, 5, 6, 7, 8, 9}, and "X" for some reason selects position 5 as its move, and so the game is played. The game described in Fig. 2.1 would have the trace shown in Table 2.1 using the notation that

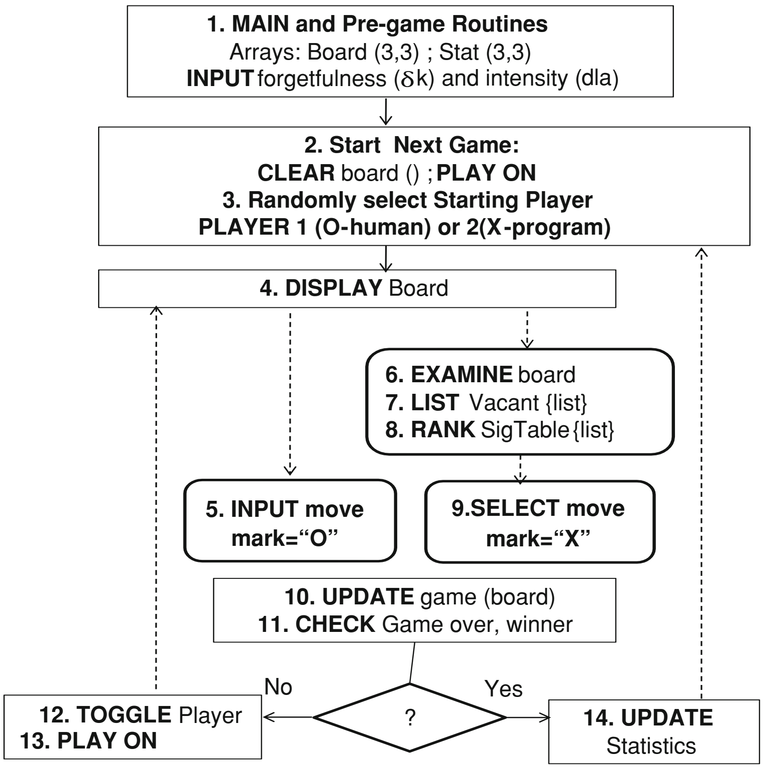


Fig. 2.2 Logic for a noughts and crosses simulator

Table 2.1 Trace of game shown in Fig. 2.1

Game trace	Comments
CLEAR BOARD (9)	Clears board to all blanks
RANDOM (player, O,X)	Choose who goes first (O in this case)
PLAY	Start of game
BOARD (1) = "O"	Move O1"—chooses top LH corner
BOARD (5) = "X"	Move X1"—selects middle
BOARD (9) = "O"	Move O2"—chooses bottom RH corner
BOARD (3) = "X"	Move X2"—selects top RH corner—mistake!
BOARD (7) = "O"	Move O3"—chooses bottom LH corner
BOARD (6) = "X"	Move X3"—good—but a hopeless position
BOARD (4) = "O"	Move O4"—wins and avoids a 4–5–6 loss
WINNER ("O")	Game over

the human *chooses* and the program *selects*. From now on, the notation adopted is that the winning board positions in Fig. 2.1 are depicted [1, 4, 7]. After each move, the game engine checks for victory or defeat. The computer program selected moves [5,3,6] in its loss to the human.

Notice that board positions 2 and 8 did not contribute directly to either the win or the loss, however, had either player chosen one of them, the game outcome may have been substantially different. In order to win, player “X” should have avoided the bad second move, but it was the talent of player “O” that eventually won the game. In reality, many games end in a tie because both players hold their own destiny within their selection of moves.

In noughts and crosses it is possible to win *regardless* of the opponents moves. It is well known that in the long run, certain squares on the board are more likely to provide a win than others. Section 2.5 shows how a payoff matrix can be formed to identify these preferred board positions based on a method of statistically encoding game performance using rewards and penalties.

2.2.2 OXO not Noughts and Crosses

In the noughts and crosses game in order to win, either player must get their own three tokens “OOO” or “XXX” in a row, column, or diagonal arrangement. A variation in the game requires that winning the game means getting either “OXO” or “XOX” as the target combination. It is a different game with different strategy. In this scenario, both players’ tokens contribute toward the eventual winner as in chess. Figure 2.3 shows a simulated “OXO” game.

Clearly the mechanics of the game is identical to that of noughts and crosses; but players must learn a whole new strategy to be successful. While it is only the win/loss determination that varies from noughts and crosses, clearly both players must leverage each other’s moves and positions. In this variation of the game, it was the move by player “X” in the upper right-hand corner that was *necessary* for player “O” to win this game. The cooperating agent makes this game similar in some aspects to chess in that it plays on the weakness of an opponent to defend against attacks.

In such games it is difficult to assess the split between the contributions of the opponent as opposed to the player. This will become evident in later chapters in which the BOXES method must determine if the task with which it was presented was too difficult and therefore not a learning opportunity. Table 2.2 is a trace of the “OXO” game in Fig. 2.3 and illustrates this well.

2.3 Incentives and Learning Velocity

In all AI systems, it is imperative for the software to seek to improve in its solution to a problem. This may be measured by how well it plays a game or some other performance metric. One method that supports this notion uses a payoff matrix which is a simplistic version of the BOXES method to be described later. Using it provides a method to create incentives to win or not lose and is driven by a numerical reward and penalty structure.

Table 2.2 Trace of game shown in Fig. 2.3

Game trace	Comments
CLEAR BOARD (9)	Clears board to all blanks
RANDOM (player, O,X)	Choose who goes first (O in this case)
PLAY ON	Start of game
BOARD (1) = "O"	Move O1"—chooses top LH corner
BOARD (7) = "X"	Move X1"—selects bottom LH corner
BOARD (4) = "O"	Move O2"—chooses LH middle
BOARD (3) = "X"	Move X2"—selects top RH corner
BOARD (5) = "O"	Move O3"—wins "XOX on diagonal
WINNER ("O")	Game over

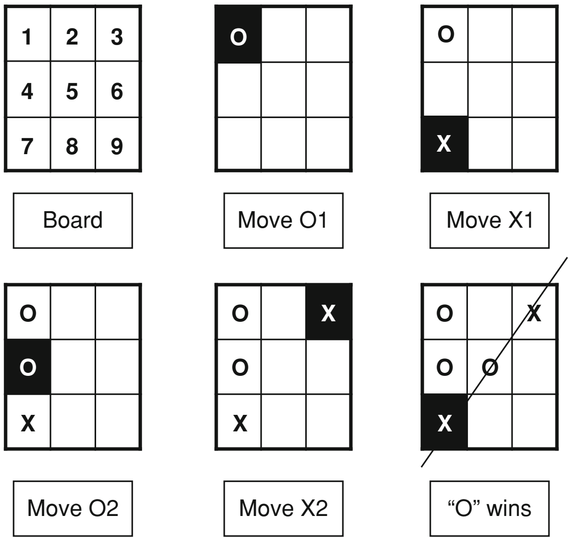


Fig. 2.3 Progression of an "OXO" game

The dilemma in writing software for games is not to over-reward the player in any particular game because the particular recent opponent might be weak or inexperienced or the player simply lucky. The litmus test is always that AI systems must play an increasingly good game regardless of the quality of the opponent.

Another problem to be handled is that if the system is to learn over time, it must process wins and losses in such a way as not to dither between values. This implies that the reward and penalty structure must be gentle enough so that the overall system can adjust its strategy based on how it performs by remembering past successes and failures. This may well create rather uninspiring rote play that is lacking in imagination and any sense of adventure.

Handling the velocity of learning must be included in the AI tautology so that drastic losses only slow down the direction in which the algorithm may be going and glorious victories merely reinforce an overall direction rather than pursuing

Draconian measures such as happens in the pruning of decision trees by genetic algorithms. The softer approach allows the algorithm to adjust to deeper or unknown facets, rules, or strategies of the game being played.

The BOXES algorithm as will be shown later includes learning coefficients that relate strongly to incentive and learning velocity. Because the initial payoff matrix is always randomly pre-seeded it is imperative that initial performances are systematically purged from the statistics using a forgetfulness factor. Also, as will be shown in Chap. 3 the BOXES algorithm not only uses the overall result of a control run to reward and penalize contributing cells, but also collects in-game data so that the responsibility (reward and penalty) for a good or poor run is distributed equitably among the contributing cells. These concepts are described in the next section using the noughts and crosses game as a model.

2.4 Design of a Noughts and Crosses Engine

Figure 2.2 showed an outline of the software structure that would be necessary for either a noughts and crosses or "OXO" program. This section describes how such an engine can be designed and implemented.

2.4.1 Overview

The game board can be represented by a set of two 3×3 arrays; one used to display the familiar graphics of the game in play and the other, which is a payoff matrix, to contain statistical data that is to be updated after every game. In the software example, for simplicity, the game board and statistical data are coded by two 9×1 strings. The reason for this simple structure will become clearer in Chap. 3 as the BOXES algorithm design is expanded upon.

During a game, when it is the program's turn to enter a mark, it must first identify all of the remaining vacant "squares" or "boxes" on the board, before it can select its move. This selection can be random if the program is to represent an idiot, or the vacant square with the highest payoff value. The move is inserted in the board array and displayed on some visual interface so the opponent (maybe a human) can enter their next move provided a win or draw has not occurred.

At the conclusion of the game, if the automaton was the winner, the statistical arrays for all positions that contributed to the win are rewarded perhaps by being increased by some numeric value and in the case of a draw or loss a penalizing scheme is applied to the same statistically rich performance arrays.

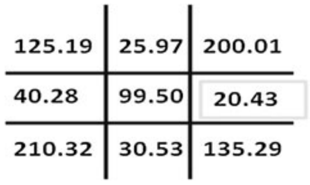
Table 2.3 Software routines in the simulator program

No	Name	Functionality/pseudo code
1	MAIN	Housekeeping in which necessary run data is input
2	CLEAR	At the start of each new game, this code clears the board array to all blanks. for i = 1 to 9 *board*(i) = blank
3	START	This randomly selects the starting player. if RND > 0.5 then *player* = 1 else *playe*r = 2
4	DISPLAY	This code displays the current game board. print *board* (1..9) in 3x3 format on a screen
5	INPUT	Allows human play using: input "Your move ";*k*: *token* = "O" if *board*(k) = blank then *board*(k) = *token* else re-prompt user
6	EXAMINE	Counts number of vacant squares *left* = 0 for *board* = 1 to 9: if *board* = blank then *left* = *left* +1
7	LIST	Inserts vacant board indices into a *list*
8	RANK	Ranks *list* positions based on *stat* data: sort (*list, stat*)
9	SELECT	This code inspects the ranked list and selects the box with the highest value. *k* = best move in list; *token*="X"
10	UPDATE game	This code inserts the last move and decrements number of remaining spaces. *board*(k) = *token; left* = *left −1*
11	CHECK	Win {OOO or XXX or OXO or XOX}, Loss {Inadvertent win by opponent}, Draw {not win or loss, and no vacant spaces} If game is over skip to #14 else #12
12	TOGGLE	Provided at least one vacant box still exists, alternate players. If *player* = 1 then *player* = 2 else *player* = 1
13	PLAY ON	Allows the next player to *choose* or *select* a move {#5 or #6) and the game proceeds
14	UPDATE Statistics	Reward and penalize statistical data based on the outcome of the last game

2.4.2 Software Sub-Structures

Table 2.3 describes the functions that appear in the program trace shown in Fig. 2.2 in more detail. This outline forms a basis for the BOXES algorithm in Chap. 3. The only difference between writing a noughts and crosses or "OXO" engine lies in variations of the **CHECK** and **UPDATE** routines. The **CHECK** function (#11) contains the algorithm that detects three-in-a-row (for noughts and crosses) or "OXO"/"XOX" and declares a draw, winner, or no decision. The **UPDATE statistics** function (#14) scans the final board and identifies the boxes that contributed to the win, loss, or draw. It then determines how to reinforce or weaken their numerical payoff values for future game play. In the "OXO" version of the game, it might be possible to include the opponent's move selections in its reckoning of position worth.

Regardless of who the winner is, at the end of each game, the program enters the **UPDATE statistics** sequence (#14) and all statistics for any contributing move are adjusted. It is the value of these statistics that is used to rank order vacant



125.19	25.97	200.01
40.28	99.50	20.43
210.32	30.53	135.29

Fig. 2.4 Example of noughts and crosses statistical results

squares in the program's **RANK** function (#8). This is similar to how the BOXES algorithm operates as will be shown in subsequent chapters.

2.4.3 Typical Results

Figure 2.4 shows results from a noughts and crosses program written by the author that played against itself thousands of times. While not scientifically infallible, the values are plausible with the corner positions that feature in most winning combinations having the most worth. Each position was initially given 100 points and aged using a 0.99 forgetfulness factor prior to being rewarded or penalized.

It is left to the reader to code his/her own version of the software and verify if the results obtained are similar.

2.5 Chance and Trial and Error

During the training phase of any learning algorithm, it is customary to use random number generators to select moves so that any initial bias is avoided. Their only function is to represent a proponent who is a true novice. In some games, chance is added to increase the level of uncertainty and excitement in the game. Monopoly™ is one of these games in which players amass wealth and who must take chances or allow passive fate to direct their strategy. Usually, a winner is declared if all other players have become insolvent or as the player with the most assets after some set time period has elapsed. In a game in which no time limits were set, it was reported [4] that one Monopoly game lasted for 70 straight days.

2.5.1 Chance and the Community Chest

In Monopoly®, the player's game pieces frequently find themselves on a board position that is not owned nor for sale rather that mandates the player to pick up a *chance* or *community chest* card. These are external insertions and have no relationship to who the player is or to how well the player is doing in amassing wealth. Blind fate, or luck, is imposed on the player who may find his or her token for no reason moved to a different location or even committed to jail. Conversely, a boon may be granted for example securing a *get out of jail free* card, or receiving a tax payment from all of the other players.

When a player's token lands upon a position that contains such a directive, no waivers are granted and the player must pick up the next card in sequence and blindly obey it and handle its consequences. As will be shown later in the book, external irrefutable influences such as boundary conditions, physical laws, safety standards, and such must override any computed control action that would create danger, present contradictions, or lead to impossible or unreasonable situations.

2.5.2 Learning with Guesswork

When a computer game reaches a situation with which it is unaccustomed, its last resort is to guess. This is performed using a random number generator to select between options. This is parallel to playing against someone who has no notion of the game rules or little experience of how the game is played. However, a random number generator does provide an excellent data source for use by a learning engine as its coverage of the state space can identify potential logical flaws and how irrationality should be handled. If the problem is a puzzle, recalling that it is a game with only one player, guessing is usually disastrous. If the problem is game oriented, mistakes arising from dealing with random inputs produce good test situations and promote scenarios for future learning improvements.

A trial and error training methodology is beneficial in that it can never become algorithmically trapped inside some local minimum and may cause the system to explore areas of the solution space that would otherwise remain uncharted.

2.5.3 Random Initialization

In a black-box controller such as that of the BOXES method, it is imperative that the system be purged of as much a priori knowledge or bias as possible. For example, a game playing algorithm cannot demand that it be allowed to start every game in order for it to be effective. As will be seen later in Chap. 4, many versions of the pole and cart control problem begin with the cart located at rest in the center of the track and the pole held vertically. On release, the system begins its motion as an inverted pendulum and the control laws take effect. But when a mathematical analysis of the pole and cart system is performed [5] it can be shown that because it is a bang–bang system there is rarely or never a steady state position where all

Fig. 2.5 Position rankings from noughts and crosses software

125.19	25.97	200.01
40.28	99.50	20.43
210.32	30.53	135.29

Board position values from software

Value rank {7, 3,1, 9, 5, 4, 8, 2, 6}

Box	Data	Rank
1	125.19	3
2	25.97	8
3	200.01	2
4	40.28	6
5	99.50	5
6	20.43	9
7	210.32	1
8	30.53	7
9	135.29	4

THE PAYOFF MATRIX

the state variables are zero. It would appear that there is folly in starting the system in an impossible initial state and expecting much meaningful learning to occur beyond handling that one particular situation.

It could be that some of the techniques used in pole balancing, which is a very different problem have inadvertently migrated to the startup procedures of the pole and cart system. As Chap. 10 mentions, a feature of chaotic systems is their sensitivity to initial conditions, so it is imperative that the control method not inherently favors any predilection in initial conditions.

The *level playing field* idiom used in today's vernacular implies that equal opportunity is a necessary part of an application, policy, or opportunity. It is a given that either player be not given any advantage or be allowed to cheat. During testing, however, it is not uncommon to create scenarios that are far more difficult to handle than are expected to occur in real life. An example of this is found in the robot-and-the-box described in Sect. 1.1.3 where technicians moved the box to be pushed or suddenly closed a door mid-plan in order to see if the robot could detect the problem and replan, rather like how modern day automotive navigation systems recalculate routes when a driver misses a turn.

2.5.4 Positional Move Strengths

In order for a program to become facile in any game, there has to be some method or metric that describes the worth or value of each token and the location of where it is or might be placed. In chess, the former is most important. No chess player would normally give up a queen to secure a pawn. The spacial rearrangement of the pieces on the game board is governed by the rules of play. Moves vary and are based on the advantages that each different chess piece affords to an overall strategy, and the guile of the player moving it. In a game such as noughts and crosses each token has equal value and the outcome of the game hinges exclusively on board positions.

Table 2.4 Trace of game using the payoff matrix

Game trace	Comments
CLEAR BOARD (9)	Clears board to all blanks
RANDOM (player, O,X)	Choose who goes first (O in this case)
PLAY ON/BOARD(1) = ”	Move O1”—chooses top LH corner
PAYOFF MATRIX ranked list is now {7, 3, 9, 5, 4, 8, 2, 6}—“1” missing	
BOARD (7) = “X	Move X1”—selects bottom LH corner
BOARD (9) = “O”	Move O2”—chooses bottom RH corner
PAYOFF MATRIX ranked list is now {3, 5, 4, 8, 2, 6}—“1, 7, 9” missing	
BOARD (3) = “X”	Move X2—selects top RH corner—mistake!
BOARD (5) = ”O”	Move O4—wins and avoids a 3-5-7 loss
WINNER (“O”)	Game over

For a metric to be effective, it must record information about sequences of moves and their outcomes, and then present options to assist future bidding for selection by the game player. In addition the metric must include an algorithm that not only takes into account the most recent game but also weighs it against prior experience to avoid unacceptable levels of indecision.

Section 2.6 is an illustration of such a method that uses what is called a *payoff matrix*.

2.6 The Payoff Matrix

Figure 2.4 showed a set of statistical data produced by one of the author’s programs and actually does reflect some measure of the positional value of each board position. This is called a payoff matrix and is illustrated in Fig. 2.5.

Using the payoff matrix, a game might follow the game trace shown in Table 2.4 which is depicted in the board shown in Fig. 2.6.

While obviously not actually aware of the status of the game at any point, the payoff matrix selects positions that are statistically preferred in the winning process. While still a loss, “X” was at least trying to win across the diagonal [3,5,7] and did not really make a mistake; it was just outplayed by the human.

The trace table confirms that the program has no strategy for winning based on the particular game board at any time. It just suggests positions that have historically been advantageous as indicated by the payoff values. In essence this algorithm represents a black-box noughts and crosses player whose board selections are purely data driven. The payoff values of the board positions are upgraded after a successful game or reduced after a loss or draw. The increments and decrements that are applied to the payoff matrix values reflect adjustable levels of desire to win or loss avoidance and can be altered by the programer at will.

The BOXES method uses a variation of the payoff matrix approach called a signature table.

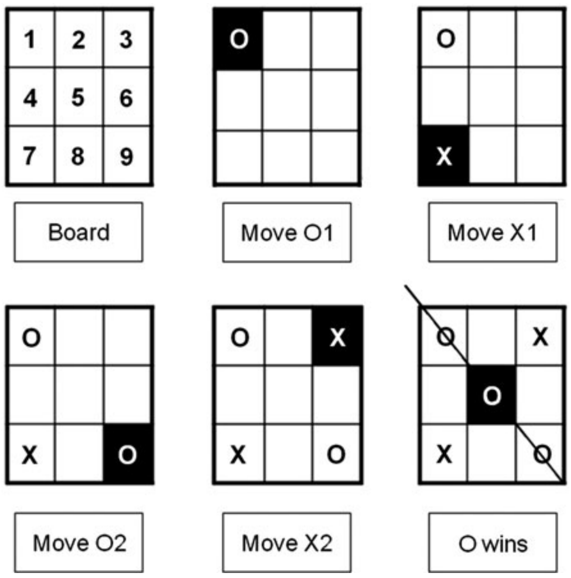


Fig. 2.6 Noughts and crosses using a payoff matrix as one player

2.7 The Signature Table

The payoff matrix method ranks relative board positions in the computer games that have been discussed in this chapter. The prevailing element of these games is the selection of a location upon which to place a single token. At no time can the token be changed during play. Referencing chess, for example it is apparent that there can be a variety of tokens that the player can select to place on any specific square during a game. In this case, the problem is now two-fold; the system must decide which square should be used next and what token should be moved there. When a payoff matrix has more than one decision to handle, it becomes multi-dimensional and it is called a signature table. In this context it would be now necessary that every board position in the game has a payoff matrix value for each token that might occupy that position. Obviously the selection of the move and token becomes much more complicated.

As before, at the conclusion of such games the statistics for all positions that contributed to a favorable outcome are rewarded by being increased by some numeric value that is based on the magnitude of the win and the intensity or desired level of achievement (called *dla* in Fig. 2.2) in winning. In the case of a draw or loss, a similar reward and penalty scheme is applied to the signature table values. If the data is over rewarded for winning, it is not necessarily creating a prudent, vis-à-vis nonlosing, style of play.

Because the signature table is seeded randomly initially, devastating bad decisions may unknowingly be present in it that surface especially near the beginning of a learning run. Therefore during the learning phase, a forgetfulness factor (called δk in Fig. 2.2) is used to downplay earlier data in the signature table as the system matures.

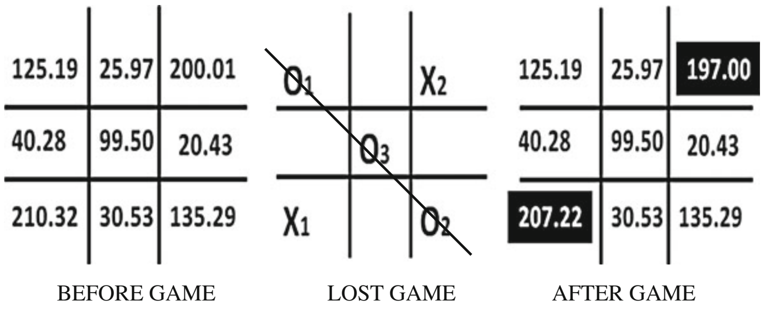


Fig. 2.7 Update of payoff matrix after a loss

2.8 Rewards and Penalties

In the payoff matrix and signature table structures, each data cell is updated at the conclusion of the game using its utility in the last game using a reward and penalty algorithm. For example, each cell might be given some random initial value say between 1 and 100. Then in noughts and crosses, the data value in a cell that featured in the last game might be aged by multiplying its previous data value by δk (perhaps equal to 0.99) and then incremented by one for a win, decremented by one for a loss, and left otherwise unaltered in the case of a draw.

After playing the game described in Table 2.4, which resulted in a loss, the payoff matrix would be updated according to Fig. 2.7, which shows the payoff values of cells 3 and 7 reduced because of their contribution to the lost game.

While the payoff values in cells 3 and 7 still contain dominating values after this one game, repeated losing play that includes these two cells will eventually drive their payoff values below, for example, those in cells 1 and 9 and alter future game play. As described in Sect. 2.3, this reward and penalty method can be adjusted to become an aggressive win seeker by setting the win increment and the loss decrement to a number greater than unity. This is called intensity or desired level of achievement (*dla*) in Fig. 2.2. The statistics will now change drastically after each game and create a payoff structure that would play a *devil may care* game with no regard for defensive play, or conversely play with a cautious defensive minded attitude.

One flaw in this method of course is that the program has no notion of how experienced each cell is and therefore initially plays randomly. In the BOXES algorithm, the signature table must contain enough data to reflect not just how valuable a cell, or box, token is, but also how certain the system is of its decision. This entails keeping track of how often that cell has been entered and what affect it had on the system outcome. The signature table method is sometimes called *learning with a tutor.*

2.9 Failure Driven Learning

In most games featuring at least two players there is a common thread that relates to how the software causes learning to occur. This is the notion of learning through failure. If success is the only measure of attainment then it is very likely that other players are better but nobody would know.

To really improve, a chess player must not only beat all comers but also succumb to losses sustained by engaging with better players or by his/her own mistakes made during a game. As this book progresses toward applications of the BOXES method to the control of unstable mechanical systems, it is the failure of the system that drives the learning process. Failure is not always a negative. The author [6] reflected on the different perspectives on failure citing that it can be constructive, limited, and good for training. It is suggested that it is good to lose sometimes in order to attain better long-term performance.

2.10 Concluding Thoughts

The BOXES method differs appreciably from the TABU [7] search genre in which past solutions are marked as unavailable so that other solution trajectories are possibly located. Rather, the BOXES algorithm enters the same states repeatedly and learns to assess and adjust their contribution over many runs over time. Because the BOXES signature table is initially randomly seeded it is to be expected that the control data in each cell will flip between values fairly often, particularly in the early learning phase of operation.

In the board games discussed above, the integrity of each player's move is maintained so that the game board progressively fills with the player's tokens until no moves are left when a draw is declared. Chess has an opposite mechanism because the pieces eliminate each other from play, with the exception of the kings of course, usually producing a sparser and sparser scenario as the game progresses. Chess pieces are assigned different values and are limited in their mobility. If a pawn gets all the way across the board to reach the opponent's edge, it is promoted to any other piece (usually a queen) except a king or pawn. The new piece is placed where the pawn ended its movement. In normal play, the density of pieces on the board decreases as the game progresses until a checkmate or draw occurs. A draw is only declared when both players see their winning cause as hopeless which is usually brought about by one player being able to endlessly move out of check, when insufficient resources are available to end the game, or when an agreed upon maximum number of moves has been made. Sometimes a draw is offered when one player foresees that there is no eventual end to the game even though the current board scenario may not overtly indicate that situation.

In most of the board games considered so far, the token values themselves are unchangeable. Players are assigned an "O" or an "X" and these are immutable during play. In chess, each side has the same 16 pieces varying only in color, with the exception of a pawn reaching the far end of the board where it is crowned and transformed into another token such as a queen. At least one game allows tokens to be flipped in value during play as will now follow.

2.10.1 Reversi (Othello®)

Reversi [8] popularly known as Othello is a game in which the actions of one player cause the actual ownership (vis-à-vis color) of the game pieces to flip. This suggests that the actual location on the board is not as important as the relative positioning of the game pieces to each other. Reversi is an abstract strategy board game that focuses on minimizing luck. It is similar to chess and noughts and crosses in that there are no hidden non-deterministic elements (such as the shuffled cards in Monopoly) and the two players, who each own one particular token color take a finite number of alternating turns until the board is full or one player owns all of the tokens. Players lose a turn if they are unable to reverse at least one of their opponent's pieces. The final winner is the player with the greatest disk count inventory or when there are no legal moves left to be played by either player. While mathematically speaking Othello still remains unsolved, the game can be programed fairly easily. It is the notion that any given board position can alter its token value during the game that makes it of interest in the BOXES method which is the topic of the remainder of this book.

2.10.2 The BOXES Method as a Game

An understanding of the above concepts is very important as the reader proceeds to Chap. 3. The BOXES algorithm takes advantage of many of the above methodologies and it is only the definition of the *board*, the *tokens*, *rules*, and *strategies* that are different from a board game. Rather than winning a game, in the real time control application, the object is rather to aggressively learn to prolong the game so the system remains in control. The data associated with each system integer is aged over time so that past performance is progressively forgotten, and contributing states are rewarded or penalized based on the outcome of each control run. Based on this data, any given signature table entry can be flipped between legal values during a set of control runs.

References

1. Michie D (1986) On Machine Intelligence. 2nd ed. Ellis Horwood: 24
2. Hayes-Roth F (1985) Rule based systems. *Communications of the ACM Vol. 28(9): 921–932*
3. Lamport L et al. (1982) The Byzantine Generals Problem *ACM Transactions on Programming Languages and Systems* Vol. 4(3):382–401

4. Monopoly 75 years young (2010) http://www.hasbro.com/monopoly/en_US/discover/75-Years-Young.cfm (Accessed 6 January 2011)
5. Rees SJ (1978) An Investigation into Possible Applications of Learning Control Systems. PhD Thesis: Council for National Academic Awards. London
6. Russell DW (1994) Failure Driven Learning in the Control of Ill-defined Continuous Systems. *Cybernetics and Systems*. Vol. 25(4):555–566
7. Glover F. and Laguna M (2001) Tabu Search. Kluwer Acad. Publ., 2001.
8. Reversi. http://en.wikipedia.org/wiki/Reversi (Accessed 3 January 2011)

Chapter 3
Introduction to BOXES Control

3.1 Matchboxes

The following is based on a Wikipedia [1] listing for Donald Michie:

> In 1960, he (Michie) developed the Machine Educable Noughts And Crosses Engine (MENACE), one of the first programs capable of learning to play a perfect game of Tic-Tac-Toe. Since computers were not readily available at this time, Michie implemented his program with about 300 matchboxes, each representing a unique board state. Each matchbox was filled with colored beads, each representing a different move in that board state. The quantity of a color indicated the "certainty" that playing the corresponding move would lead to a win. The program was trained by playing hundreds of games and updating the quantities of beads in each matchbox depending on the outcome of each game.

Rather than construct arrays of matchboxes each with its own sophisticated labeling and bead shuffling schema, computer programs can be written to perform similar data-driven procedures to those illustrated as the payoff matrix to simulate the colored beads. MENACE was certainly a prototype of what Chap. 2 defined as a signature table and the matchbox method title became abbreviated to BOXES.

It is the transition of this concept to a real-time system that is of special interest to the reader. In order to move away from board games to focus on real-time control, a paradigm shift is necessary. Instead of the *player* selecting a position on the board, the real-time *system* provides state variable values that identify the current dynamic state in which the system is operating. In other words the location of the next move is determined by the system, it is what token is to reside there that is of interest. This is equivalent to knowing the state of the playing board but now the algorithm returns a value (or token) for use in direct control of the application such as a switch or motor directive.

3.2 Components of the BOXES Method

What is so different in the BOXES Method when applied to a real-time system is that, unlike in a state space approach [2] to control, very little a priori knowledge of the system under consideration is needed to achieve controllability. This type of controller is sometimes referred to as *black box* [3] control and uses what is known as a *signature table* to hold control decisions. A software structure that supports this was introduced in Sect. 2.6.

In order to understand the fundamental concepts of any BOXES method system, it is necessary to define the game board; identify game pieces; establish how to select legal values for those pieces; create an end game detection mechanism; and finally, enforce strategies for optimality and learning.

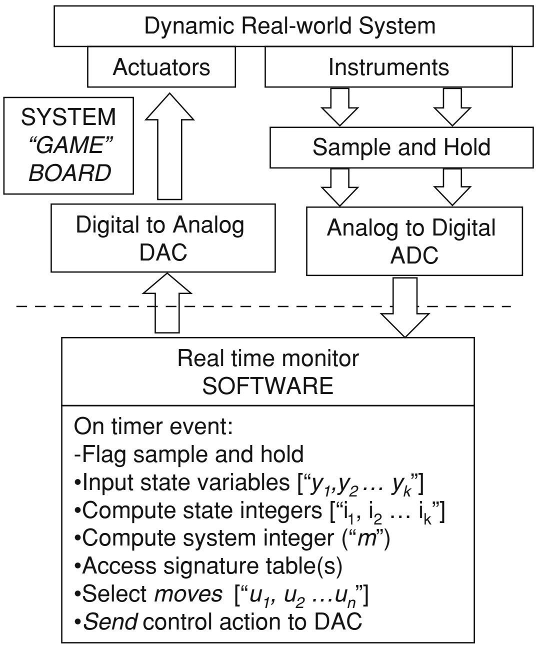
3.2.1 Defining the Game Board

Black box controllers unlike state space controllers require very little a priori knowledge of the actual dynamic system under consideration to achieve acceptable levels of control. Readers who are familiar with the control of dynamic systems will hardly need reminding of how they are described by blending a mathematical model with real state variables. Data from the real world are correlated with and compared to models of the variables and a control strategy enacted. For this to occur, real variables such as temperature, pressure, position, velocity are converted into electric signals for input to a digital controller. The mechanisms for measuring, sampling, and converting real-world data into digital systems are well known, as are the processes for returning digitally computed control values to the system using some actuation mechanism.

The process must be carefully orchestrated timewise. It must be slow enough to allow the conversion of analog signals to digital data and for the BOXES computations time to process. The control results must then be converted from digital to analog form, amplified, and applied to the system at regular intervals. On the other hand, it must be fast enough to be effective in controlling variances in the system variables. For this to occur, it is customary that the sample rate be about 1/20 of the system's time constant and varies from application to application. The system time constant is an area where some a priori knowledge is required.

Instead of a board or sheet of paper, this control *game* is played by processing state variables that are the sampled data of the system, and returning appropriate control moves that are to be learned over time. Initially, the BOXES system returns control values somewhat randomly and seemingly uses trial and error and guesswork as its primary strategy. However, the BOXES algorithm is designed to use data collected over time to learn to *play* better and better games until it acquires great skill. Figure 3.1 illustrates this process.

Fig. 3.1 The dynamic system game board


Dynamic Real-world System
Actuators
Instruments
SYSTEM "GAME" BOARD
Sample and Hold
Digital to Analog DAC
Analog to Digital ADC
Real time monitor SOFTWARE
On timer event:
-Flag sample and hold
•Input state variables ["y_1,y_2… y_k"]
•Compute state integers ["i_1, i_2 … i_k"]
•Compute system integer ("m")
•Access signature table(s)
•Select *moves* ["u_1, u_2 …u_n"]
•*Send* control action to DAC

3.2.2 Identifying Game Situations

The output from the Analog to Digital Convertor (ADC) system populates the software with data corresponding to what are called *state variables* at any given sample time. The BOXES methodology requires that each state variable become a *state integer* in much the same way that truth values in fuzzy logic[*n*] are determined. Rather than using a nomenclature such as Very Hot, Hot, Warm, Cold, and Very Cold, the BOXES method simply requires that all variables are classified by a zone integer in which they each currently exist. Figure 3.2 shows an example for some hypothetical state variable $y(t)$ being processed in this case into one of three zones as defined by the boundary values b_1 and b_2. It should be noted that these boundaries are not necessarily equally spaced or that they be even or odd in number.

The *state integer* identifies the region of operation of each variable at any given time and is implied as so throughout the remainder of this book. A system may have several state variables, in which case a state integer must be assigned to each in the fashion described above. As will be shown, once the number of subdivisions for each variable is established, it is quite simple to group these integers into one *system integer.* In order for these integers to be combined to form a unique system integer it is necessary only to know how many possible zones exist for each variable.

Equation 3.1 shows this calculation for a system with "k" state *variables* [y_1, y_2 … y_k] all of which have been processed into "k" state *integers* [i_1, i_2 … i_k] as

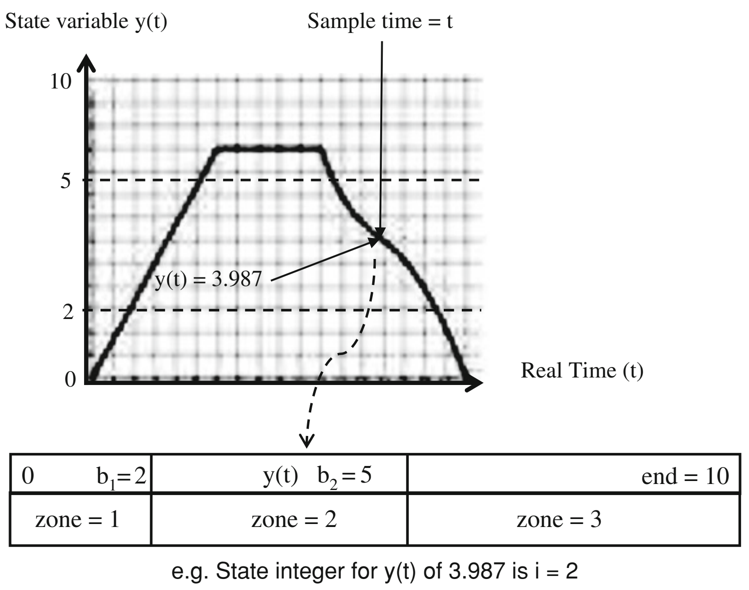


Fig. 3.2 Example of a state integer "*i*" for state variable *y*(*t*)

shown above. To do this it is necessary to know into how many regions each variable could be divided into and this is denoted as [Nb$_1$, Nb$_2$ … Nb$_k$].

$$m = i_1 + \mathrm{Nb_1}^*(i_2 - 1) + \cdots \mathrm{Nb_1}^*\mathrm{Nb_2}^* \cdots^* \mathrm{Nb}_k{}^*(i_k - 1) \qquad (3.1)$$

The system integer always lies in the range given by the product of the number of boundaries into which each variable can be divided, as shown in Eq. 3.2.

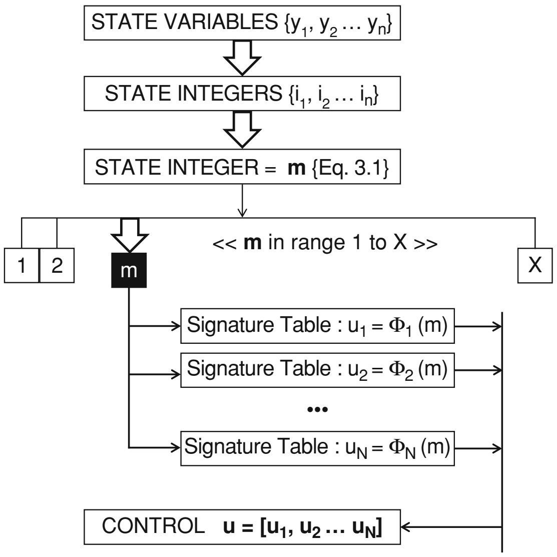
$$\mathrm{range}(m) = 1 \to \prod_{j=1}^{k} \mathrm{Nb}_j \qquad (3.2)$$

The zones into which each variable is apportioned need not be equally spaced across the variables and can be an odd or even number.

For example, if there are four state variables then a set of four integers [i_1, i_2, i_3, i_4] would be needed to define the current (dynamic) state of the system to be controlled as shown in Eq. 3.3, where Nb$_1$, Nb$_2$, Nb$_3$, and Nb$_4$ are the number of zones in each of the four state variables.

$$m = i_1 + \mathrm{Nb_1}^*(i_2 - 1) + \mathrm{Nb_1}^*\mathrm{Nb_2} + (i_3 - 1) + \mathrm{Nb_1}^*\mathrm{Nb_2}^*\mathrm{Nb_3}^*(i_4 - 1) \qquad (3.3)$$

This implies that all four variables can be assigned a state integer in the range 1–5, and the legal values of system integer (m) would lie be between 1and 625. This translates to a game "board" with 625 legal "squares" from which to choose.



STATE VARIABLES {$y_1, y_2 \ldots y_n$}
STATE INTEGERS {$i_1, i_2 \ldots i_n$}
STATE INTEGER = **m** {Eq. 3.1}
1
2
m
<< **m** in range 1 to X >>
X
Signature Table : $u_1 = \Phi_1$ (m)
Signature Table : $u_2 = \Phi_2$ (m)
•••
Signature Table : $u_N = \Phi_N$ (m)
CONTROL **u = [$u_1, u_2 \ldots u_N$]**

Fig. 3.3 Reading control values from signature tables

If ever a variable falls outside of its allowable minimum or maximum range, its state integer is set to zero and the BOXES algorithm sets the system integer to zero thus denoting failure in the control process. At this juncture, the BOXES algorithm completes any partial calculations for the last valid cell and asserts an end of run status.

3.2.3 Selecting Game Piece Actions

As stated above, once all state integers are known, a system integer (m) can be easily computed and is used as the index to point to a corresponding entry in the signature table. In some applications more than one signature table may exist for each pointer value. In that case, each signature table acts as a payoff matrix and returns a set of control values—called u in Fig. 3.1. Actuators connect this to the real system and the process repeats at some appropriate sampling frequency. Figure 3.3 illustrates the mechanism for multiple signature tables.

The control set $u = [u_1, u_2 \ldots u_N]$ in Fig. 3.3 is conditioned for system power requirements and attached to the system's motor control centers, ON/OFF switches, or other actuation mechanisms. Access is granted to non- zero system integers produced within the range of the system's variables and in fact as stated above it is only when the "player" wanders off the board that the system integer gets set to zero that the *game* ends!

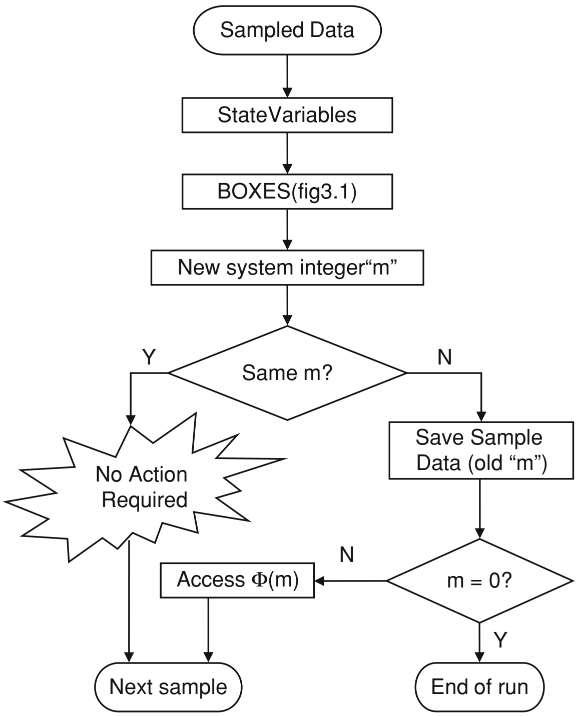


Fig. 3.4 Control and data collection using the system integer

3.2.4 Real-Time Data Handling

In order to be able to track which control regions are being used during the control run, it is important not only to track the system integers but also to keep temporary data on how often any particular "box" is entered and some measure of how long the system remains in that state.

If many cells are fleetingly entered it is fairly certain that the system is thrashing and probably not performing very well. If only a small number of states are rapidly entered many times, the system may well be getting locked in a local minimum and quivering across one of the state boundaries. If a repeating pattern of states with fairly lengthy durations in each cell is noticeable, the system is performing some form of controlled oscillations. The task of the BOXES software is to promote this latter case.

If the software is designed to keep this non-permanent data during a control run, the update of the more permanent performance data occurs only after a failure when the system is not in *reactive* mode. Of course, if the state variables are being sampled 50 times every second it is not possible to keep data on every sample, but rather whenever the system leaves one cell and enters another. In Chap. 4 a specific example explains how this seemingly enormous volume of data can be fused into data structures that are easily indexed by the system integer. Figure 3.4 illustrates how if the system being controlled does not change state, the current control actions are unaltered. On changing state, the algorithm saves data for the state being exited rather than sample by sample. In this manner the data structures for real-time data are bounded only by the number of possible values of the system integer as explained above. With some ingenuity, this data can be used to categorize the significance of the contribution of any cell to the current run when it is over.

3.2.5 Detecting an End Game Situation

Should any state variable drift outside of its pre-allocated minimum or maximum values, the system is deemed to have failed and the value of the offending state integer is set to zero forcing the system integer to be set likewise. This flags the BOXES algorithm to stop acting in the real-time *reflex* or *reactive* control mode and enter a learning and *reflective* phase.

In reflective mode the system examines data from the previous run. As will be explained in Sect. 3.3 real-time data is accrued during each run for the system as a whole and for the contributing states. It is in this phase that statistical data is updated and any necessary adjustments made to the signature table values based on a penalty and reward system before another is run authorized. This is the simplest form of the two phase BOXES operation.

3.3 Updating the Signature Table

Once a control game is deemed over, as indicated by a zero system integer value, the BOXES algorithm must compute metrics that first give an overall sense of increased or decreased achievement, and secondly how to reward or penalize contributing boxes. The overall performance metric varies from application to application; if the BOXES controller is attached to a pole and cart system, it is the length of the duration of the run in which the system was in control that would be used. If it is attached to a process system, the metric might be related to product yield or quality. If the controller objective is to reduce bounce or oscillations, the metric might be the integral under the state variable curves, and so on.

3.3.1 Overall Performance Data

Once the immediately previous run is over, the metric must be merged into overall historical data. If the performance metric is the last run time (Tf) the historical metric might be a weighted average run time. To ensure that this value is correct entails keeping real-time system data such as a count of the number of runs the system has performed so far and a method of computing an average global runtime over many runs. Equations 3.4–3.6 illustrate this.

$$\text{Global\_runtime} = \text{Global\_runtime}^{*}\,\delta k + \text{Tf} \tag{3.4}$$

$$\text{Global\_count} = \text{Global\_count}^{*}\,\delta k + 1 \tag{3.5}$$

$$\text{Merit} = \text{Global\_runtime}/\text{Global\_count} \tag{3.6}$$

where

Tf	the time to fail in the previous run
Global_runtime	the aged total time of all runs
Global_count	the aged experience of the system
δk	a forgetfulness factor that down rates prior runs

This value of *merit*, which is really a weighted average runtime can be used as an overall performance metric as it reflects the control performance of the algorithm over long periods of time. Should the system be tasked with reducing oscillations or with some other control problems, another value of merit would be appropriately constructed. The corollary of BOXES *merit* in the game world is the measurement of a player's *skill* level.

3.3.2 Desired Level of Achievement

Once a value for the system merit has been established, it is advantageous to try to compute a factor (dla) by which the algorithm can reward or punish the indices of the moves that contributed to the last run. Equation 3.7 is an example of this calculation:

$$\text{dla} = C_0 + C_1{}^{*}\text{merit} \tag{3.7}$$

where

C_0, C_1	coefficients of learning rate

The values of C_0 and C_1 serve to reflect a lenient or urgent learning curve. Of course, a faster learning curve may not cause the system to perform as well dynamically as a slower more experienced approach might.

3.3.3 Individual Box Decision Data

Upon the end of a run occurring, it is necessary to examine which boxes or cells were entered and how long the system remained within that control space during the prior run. The goal is to create a submetric for each cell that will measure the *strength* of each possible decision for each signature table. It is to be expected that these submetrics will be based upon the number of times the cell was entered for each decision and some measure of accomplishment that will be related to its contribution to the overall system merit.

As an example, one such algorithm for a system that has one signature table with three possible control values (u) for each cell will be considered. The three values might be related to the motor controls forward, neutral, and reverse which are designated "**F**", "**N**", or "**R**" in this example. It is assumed that new values of *merit* and *dla* have been calculated using Eqs. 3.4–3.7. For each value of m, the signature table currently contains values of either "**F**", "**N**" , or "**R**" and during the run just ended, several of those cells were entered as evidenced by non-zero values of collected data which are called *count* (m) and *time* (m) reflecting how many times each cell was entered and how long the system stayed in that cell.

Three sets of historical data for each cell are maintained designated Global_count F, Global_count N, and Global_count R reflecting the experience level of each decision for given values of m. Global_time F, Global_time N, and Global_time R indicate the total time spent in that cell when any of the three decisions are invoked. During the update phase, the recently acquired in-run data must be appended to the appropriate global data field as shown in the following pseudo-code:

```
FOR ALL values of m
IF $\Phi$(m) = ''F'' THEN
    Global_count_F(m) = Global_count_F(m)*$\delta$k + count(m)
    Global_time_F(m) = Global_time_F(m)*$\delta$k + time(m)
IF $\Phi$(m) = ''N'' THEN
    Global_entry_N(m) = Global_count_N(m)*$\delta$k + count(m)
    Global_time_N(m) = Global_time_N(m)*$\delta$k + time(m)
IF $\Phi$(m) = ''R'' THEN
    Global_entry_R(m) = Global_count_R(m)*$\delta$k + count(m)
    Global_time_R(m) = Global_time_R(m)*$\delta$k + time(m)
```

In this manner, only the historical data associated with the current signature table value is appended with the data from the previous run. If the real-time data for an unentered cell was reset to zero prior to every run, it is harmless to sweep through the entire signature table. If computational speed is an issue, a simple software change can be made so that only cells that have non-zero real-time data are selected.

Equations 3.8–3.10 illustrate how the strength of each decision for any cell (m) may be calculated. It should be noticed that this BOXES algorithm uses the global data values computed as shown above and the system merit and desired level of achievement to calculate values for the strength of each cell for each of its three possible values (S_F, S_N, and S_R). This is a measure of the strength of a decision in any individual cell. "K" is an arbitrary accelerator factor.

$$S_F = \frac{(\text{Global\_time\_}F(m) + K^{*}\text{dla})}{(\text{Global\_count\_}F(m) + K)} \tag{3.8}$$

$$S_N = \frac{(\text{Global\_time\_}N(m) + K^{*}\text{dla})}{(\text{Global\_count\_}N(m) + K)} \tag{3.9}$$

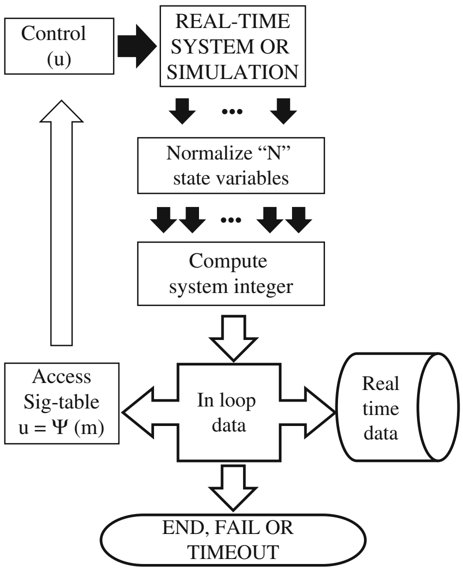

Control
(u)
REAL-TIME SYSTEM OR SIMULATION
Normalize "N" state variables
Compute system integer
Access Sig-table u = Ψ (m)
In loop data
Real time data
END, FAIL OR TIMEOUT

Fig. 3.5 The BOXES algorithm in reactive, real-time mode

$$S_R = \frac{(\text{Global\_time\_}R(m) + K^{*}\text{dla})}{(\text{Global\_count\_}R(m) + K)} \tag{3.10}$$

These three values of strength are used to determine the control value to be inserted in the signature table $\Phi(m)$ in preparation for the next control run as shown in the following block of simple pseudo-code. If by chance all three values of strength are identical, the control token is left unaltered or re-randomized.

```
FOR all values of m
Compute {Fs, Ns, Rs}---using Eq. 3.8–3.10
IF Fs > Ns AND Fs > Rs THEN Φ (m) = "F"
IF Ns > Fs AND Ns > Rs THEN Φ (m) = "N"
IF Rs > Fs AND Rs > Ns THEN Φ (m) = "R"
```

3.4 Overall Software Design

The BOXES algorithm is a software agent. It can be attached to real or simulated systems as will be shown in Part 2 of this book. Figure 3.5 depicts a software architecture for the BOXES method in the reflex or reactive mode that is in play during a control run.

Figure 3.6 shows the same system but now in its reflective or learning mode. In this mode, real-time performance data is manipulated so that the decisions in the signature table can be updated prior to the initiation of the next run.

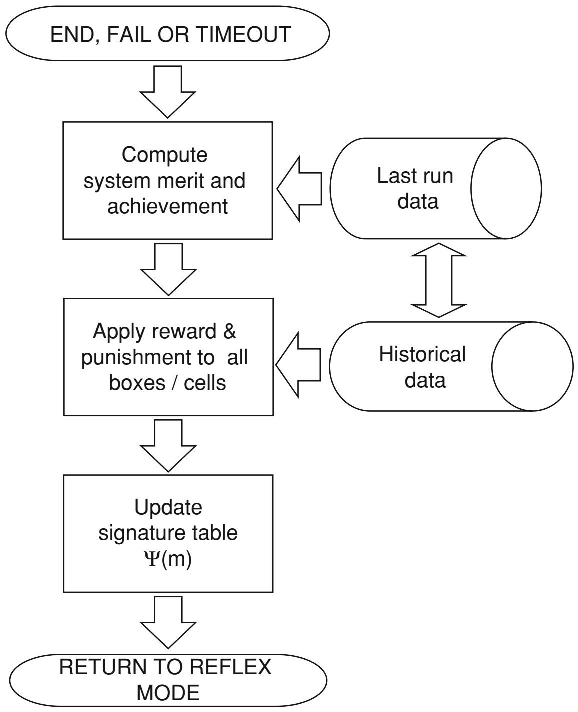

END, FAIL OR TIMEOUT
Compute system merit and achievement
Last run data
Apply reward & punishment to all boxes / cells
Historical data
Update signature table Ψ(m)
RETURN TO REFLEX MODE

Fig. 3.6 The BOXES algorithm in reflective, learning mode

When actually writing BOXES software, there are many features that require special attention that will be explained as the book progresses. Of particular interest will be how to randomly start up the system and how to assign bias-free initial values to cell data especially that implanted in the signature tables. As the system matures over many excursions of the software, it becomes apparent that the BOXES method becomes very proficient in black box control.

3.5 Concluding Comments

Within some appropriate sample time, the state variables are codified into one system integer, which is used to locate the appropriate "box" in which the system is currently residing and to select control data in a read only algorithm corresponding to that particular tuple which is then conveyed back to the system. As stated in Chap. 2, bang–bang systems are particularly suited to machine learning of the BOXES type but this is not necessarily a hard and fast rule.

Chapters 4–6 focus on one particular mechanically unstable system. Good results are produced for simulations and an actual physical rig.

References

1. http://en.wikipedia.org/wiki/Donald_Michie (Accessed September 12, 2011).
2. Williams RL and Lawrence DA. Linear State-Space Control Systems. 2007 John Wiley & Sons Inc. Hoboken, NJ.
3. Xie, C and Li, A. An Algorithm Using Genetic Programming for the Compensation of Nonlinear Distortion Based on Wiener System Model. (2004) FACTA UNIVERSITATIS (NIS)- SER: ELEC. ENERG. vol. 17: 219–229.

Chapter 4
Dynamic Control as a Game

4.1 Control of Dynamic Systems

Dynamic systems occur everywhere. They are found in vehicle controllers [1], chemical process plants [2], railways [3], and in many more situations. The three most salient features that exist in such applications are that there is a need for feedback control to ensure appropriate response times, safety, and real-time scheduling. In most systems this is accomplished using analog or digital controllers that measure the current values of system, or state, variables (length, height, pressure etc.), and in response construct some form of feedback signal based on those values. In a digital controller, there is also the need to sample the data often enough to identify variations but slow enough to allow the control action to be effective. Modern digital controllers use a state space methodology [4] in which data values are fitted to sets of differential equations that are derived from a linearised theoretical model of the system and an appropriate control signal computed. This can often be achieved within a reasonable sample time.

The BOXES algorithm is different from the state space method in three respects. First, no analytical model of the plant under control is necessary beyond some simple knowledge of the number of system variables that exist and their minimum and maximum expected values. Second, it plays dynamic control as a game during which control moves are exerted. Third, the system under control creates the board position based on real-time data, and the signature table responds with an action in much the same manner as the community chance mechanism in Monopoly®. As stated previously, to accomplish this, the BOXES methodology must define a system game board, identify game pieces, establish legal values for those pieces, create an end game detection mechanism, and finally, enforce strategies for improvement of play. A perfect game corresponds to one that is optimal.

Winning in this context is interpreted as being the situation that the system was controlled for some set interval of time.

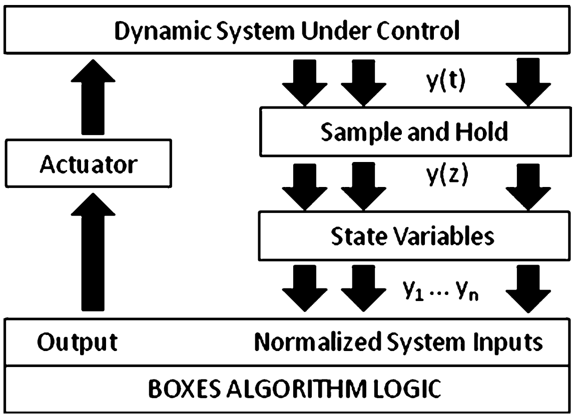


Fig. 4.1 Attachment of the BOXES algorithm to a real system

4.1.1 The Dynamic System Game Board

The BOXES method is a black box control [5] mechanism that, unlike state space controllers, requires very little a priori knowledge of the actual dynamic system under consideration to achieve acceptable levels of control. The system's real-time variables are captured using conventional instrumentation with an appropriate sample-and-hold feature. For the BOXES method to be more generic, it was found advantageous to normalize all state variables using their known minimum and maximum values as shown in Eq. 4.1.

$$y_{i\text{norm}} = \frac{(y_i - y_{\min i})}{(y_{\max i} - y_{\min i})} \tag{4.1}$$

where, for all state variables:

y_i	raw sample of the ith state variable
$y_{\max i}$	maximum allowable value of ith variable
$y_{\min i}$	minimum allowable value of ith variable
$y_{i\text{norm}}$	normalized value of the ith variable: $0 > \hat{y}_i < 1$

While the minimum and maximum values of each state variable do seem to represent a priori knowledge about the system, they really only relate to the system's limiting values, and do not contribute to its dynamic performance during a controlled run.

Figure 4.1 shows how the controller is attached to the real world in much the same way that a board game requires a move by move display.

These normalized values represent the current state of the dynamic system and all exist in the interval zero and one. As the system proceeds in its dynamic motion, of course, these values may alter with every sample. In the BOXES controller it is never the absolute values that are important, rather the values in which zone each state variable currently exists. This is similar in some regard as to

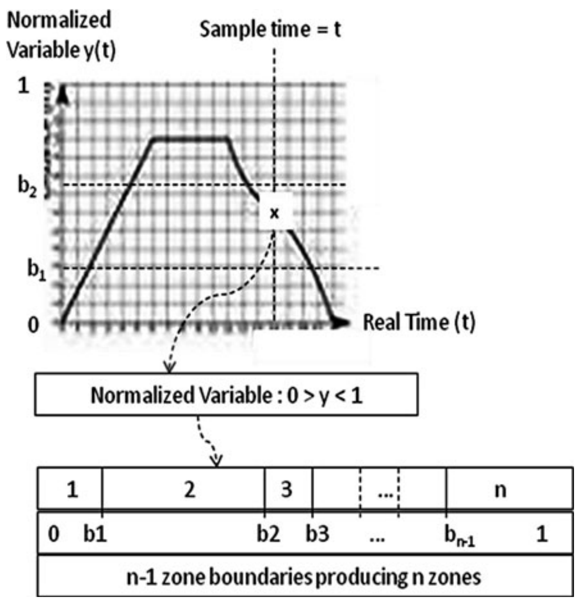


Fig. 4.2 Conversion of a state variable to a state integer

how fuzzy logic systems are designed where, for example, a temperature value is classified as very cold, cold, lukewarm, warm, hot, or very hot. Fuzzy classification uses boundary points to delineate at what point variables switch between the fuzzy regions. Fuzzy logic systems do allow membership in more than one region at any time, whereas in the BOXES algorithm, the regions in which each variable exists are crisp and used to compute a crisp system integer that completely identifies the state of the system at any instant of time.

An introductory example of this was given in Sect. 3.2.3, and a more in depth example now follows in Sect. 4.1.2 and in Chap. 5, where the state variables of a real physical system are encoded as part of a trolley and pole project.

4.1.2 State Variables and State Integers

Readers who are familiar with the control of dynamic systems will hardly need reminding of how systems are described using state variables. Figure 4.2 illustrates the mechanics of the process of translating a state variable into a state integer.

After normalization, each state variable always exists in one of the regions that are produced by dividing the range into finite zones by a set of boundary values which for now are assumed fixed during the lifetime of any configuration of the BOXES system for each state variable. These zone boundaries exist for each state variable as a set of different, not necessarily equal, decimal values in the zero to one range. For example, in Fig. 4.2 the two boundary values might be 0.5, and 0.9 which would create three possible integer zones in which the variable might exist.

These boundary values may also be considered to constitute some level of a priori knowledge. Chapter 13 of the book will examine how the values of these boundaries can be programed to evolve within the range based on the strength or resolve of the cell strengths developed during many control runs. Of course, the number and spacing of these boundaries has been investigated by many researchers, for example, [6], in this field over the years. Some have conjectured that there should be many, equally spaced zone boundaries for each state variable. Others state that the boundaries should located at or near the 1-, 2-, and 3-sigma points in the range of the variables. These and other combinations of ideas all have merit. For the present, it will be assumed that the actual number and location of all of the zone boundaries are unimportant.

To enable the BOXES signature table to be accessible to the whole system, normalized state variables must all be converted into state integers and an overall system integer computed. This one value becomes the index from which all BOXES manipulations are keyed. In game terms, it is the next move!

4.1.3 Creating a System Integer

Figure 4.3 shows an example of a real world process that has two state variables $\mathbf{y_1}$ and $\mathbf{y_2}$. At some sample time (t) their current values are sampled and an algorithm applied that locates them within one of their appropriate zones or regions. In the example shown, the first variable is identified as being in region 4 out of a possible 5, and the second variable is in region 2 out of a possible 3. These regions are called *state integers* and are true only within that particular time interval. If the system is dynamically slow, one or more of the state integers may persist in the same zone over several samples.

Having identified the two state integers, it is a simple matter to combine them into one unique system integer, which is usually denoted as *m*. The only factors that are necessary for this calculation (see Eq. 4.2) at this point relate to the number of sub-divisions (shown as $\mathbf{Nb_1}$ and $\mathbf{Nb_2}$ in Fig. 4.3) in each state variable as indicated above.

$$m = i_1 + \mathrm{Nb}_1 * (i_2 - 1) \tag{4.2}$$

For the example, $i_1 = 4$, $i_2 = 2$ and $\mathrm{Nb}_1 = 5$. The second number of boundary values ($\mathrm{Nb}_2 = 3$) but does not feature in the calculation giving a system integer value for the data shown of $m = 9$. For this system there are only fifteen possible system integers, consequently this particular *control game board* would be described by a 15×1 array in the same way that the noughts and crosses board was coded as a 9×1 array.

As will be shown, once the number of sub-divisions for many variables is established, it is quite simple to group these integers into one *system integer*—which throughout will be called *m*. Its value is bounded by the product of the number of subdivisions (boundaries) for all of the system variables. For example, if there are four state variables then a set of four integers [i_1, i_2, i_3, i_4] would be needed to define the current (dynamic) state of the system to be controlled. In order for these integers

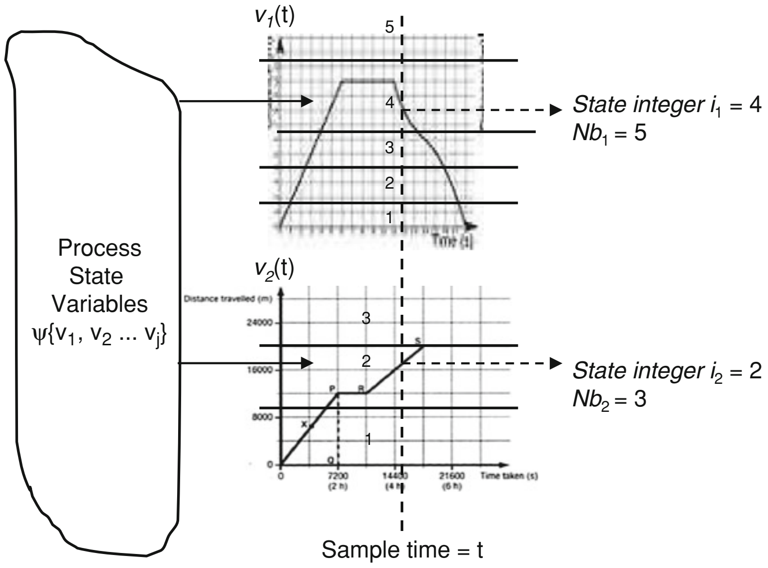


Fig. 4.3 Conversion of multiple state variables into state integers

to be combined to form a unique system integer it is necessary only to know how many possible zones exist for each variable. Eq. 4.3 shows this calculation:

$$m = i_1 + \mathrm{Nb}_1 * (i_2 - 1) + \mathrm{Nb}_1 * \mathrm{Nb}_2 * (i_3 - 1) + \mathrm{Nb}_1 * \mathrm{Nb}_2 * \mathrm{Nb}_3 * (i_4 - 1) \tag{4.3}$$

Where, Nb_1, Nb_2, Nb_3 and Nb_4 are the number of zones in each of the four state variables, respectively. The zones need not be equally spaced and their number can be odd or even. In general, if there are k normalized variables divided into Nb_1, $\mathrm{Nb}_2 \dots \mathrm{Nb}_k$ zones the general form of the system integer equation is shown as Eq. 4.4.

$$m = i_1 + \mathrm{Nb}_1 * (i_2 - 1) + \mathrm{Nb}_1 * \mathrm{Nb}_2 * (i_3 - 1) + \dots \prod_{j=1}^{j=k-1} \mathrm{Nb}_j * (i_k - 1) \tag{4.4}$$

Legal values of m lie between 1 and the product of the "k" values of Nb. If any state integer falls outside of its range, the system integer is set to zero, indicating that the game is over.

4.1.4 Signature Table Control

The value of the contents of the signature table may alter between applications, but it is normally limited to one variable with two or more values. In the trolley and pole studies to follow in Chaps. 5–7 in which the system output directly drives a

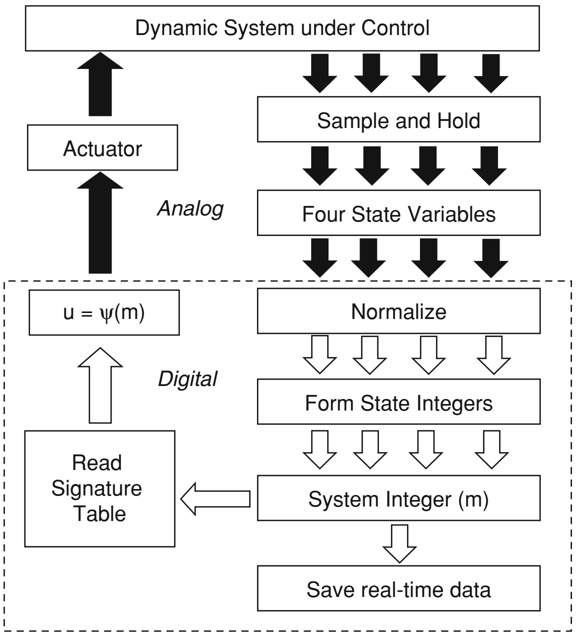


Fig. 4.4 Reading the signature table for a system integer *m*

motor, the signature table contains either a LEFT or RIGHT decision. Should a NEUTRAL gear be added the system, the signature table would contain three values for each cell. As stated in Chap. 3 what is different from the board game metaphor is that in real-time control the token value(s) for any system integer frequently change from game to game based on performance. During a control run the signature table is accessed in read-only mode and uses the current value of the system integer as the pointer or index to the array of values. This closed loop method is common in any computer controlled system with the basic difference that a conventional control algorithm normally uses derived values of the system variables themselves to form an acceptable actuation signal. Once a value has been obtained for the system integer *m*, Fig. 4.4 shows how the algorithm accesses the signature table using *m* as a pointer, and returns a real-time control response which is designated *u*.

In a black box method like this, the control value is selected from a signature table of predetermined values. During the control run, real-time data must be collected and attributed to every state that the system enters.

4.1.5 End of Game Action

Should any individual state variable drift outside of its pre-allocated minimum or maximum value, the overall system is deemed to have failed and the value of the corresponding state integer is set to zero. A zero *state integer* immediately creates a zero *system integer* which flags the algorithm that the *game* is over. At this time, the algorithm must examine all contributing states and update statistical data values in the signature table based on a penalty and reward system that will be explained in Sect. 4.3. Any corrective action deemed necessary by the update sequence must be performed rapidly if the system is to operate continuously.

4.2 Actual Real-Time Data Collection

In all prior sections when real-time data is mentioned, it has been assumed that these values are available and manipulating them is purely an algorithmic exercise. In reality, when the system under BOXES control is a physical entity, it is much more complicated and fraught with anomalies.

4.2.1 Short Duration Mechanically Unstable Systems

If the application is mechanically unstable, such in a pole and cart, there is always time for the algorithm to perform its reflective phase after a failure occurs. This can happen, for example, if the pole angle exceeds a maximum from which it can never recover, or, if the cart reaches the end of its track. In this case, if many control runs are to be performed, it is essential that the physical vehicle be restored to a position from which it can be automatically restarted. This is covered in depth in Chap. 7, but it is obvious that a human operator cannot attend the rig for long periods of time in order to manually reset the system. In a simulation, of course this is much easier to perform. In systems such as the pole and cart, the goal is usually to prolong the duration of the system knowing that a time usually comes when the system will fail. The term *failure driven learning* was coined by the author [7] and applies to this genre of controllers.

One classic performance measure is to calculate time-to-failure (Tf) for each cell. The metric computes in retrospect how often and how long after each cell was entered that the system failed. In this case, the data collection requirement is actually much simpler than that proposed in Fig. 3.4, where a record of the time of each entry and exit into any cell needed to be kept. If the metric to be used is time_to_fail (*m*) it is only necessary to keep a running sum of when any particular cell (**m**) was entered, called *sumtime*(*m*) and a count, *count*(*m*), of how often it was involved in the run. It is assumed that the control run lasted for some unit of time, called (Tf).

To illustrate this, Fig. 4.5 illustrates a situation where some cell (*m*) was entered three times, so $count(m) = 3$, at times $\mathbf{T_1}$, $\mathbf{T_2}$ and $\mathbf{T_3}$.

The real-time system would keep a record of the sum of the times when the cell was entered, called $sumtime(m) = \mathbf{T_1} + \mathbf{T_2} + \mathbf{T_3}$. The value of time_to_failure will be used to penalize the cell later. For reference, in Fig. 3.4, the block corresponding to *Save Sample Data* becomes a very simple accumulator to which each entry time is simply added every time that cell is entered. The usual time to

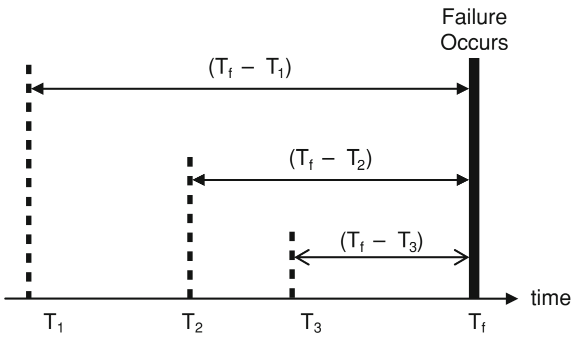


Fig. 4.5 Example of calculating time-to-failure times

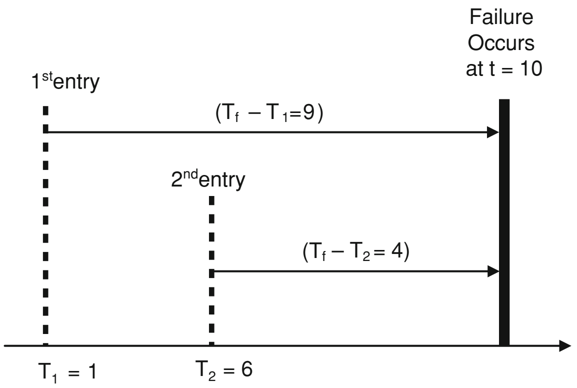


Fig. 4.6 Illustration of a cell entered only twice in a run

fail metric for cells can be calculated using Eq. 4.5 which simplifies to Eq. 4.6 so that for any cell (*m*) it is necessary only to save accumulated values for Tf, *count*(*m*) and *sumtime*(*m*) during the real-time run.

$$\text{Time\_to\_failure}(m) = \sum_{j=1}^{j=\text{count}(m)} (\text{Tf} - Tj) \tag{4.5}$$

$$\text{Time\_to\_failure}(m) = \text{count}(m) * \text{Tf} - \text{sumtime}(m) \tag{4.6}$$

The value of time_to_failure need not be saved beyond the update process but it reflects how the contribution of each cell is deemed to affect the duration of the run. To show generally how this taxonomy works, Fig. 4.6 depicts a situation in which a state was entered twice, at time 1 and 6, during a run of 10 time units. Per Eq. 4.6 its time_to_failure penalty value would be $(2 \times 10 - 7)$ or 13.

If another cell was entered in say nine equal one second intervals in the same 10 time unit run, its time_to_fail would be computed as $10 \times 9 - (0 + 1 + \cdots 9)$ or 45. This frequently entered cell obviously contributed more to the failure than the first cell as reflected by its penalty value which is greater than that of the first cell.

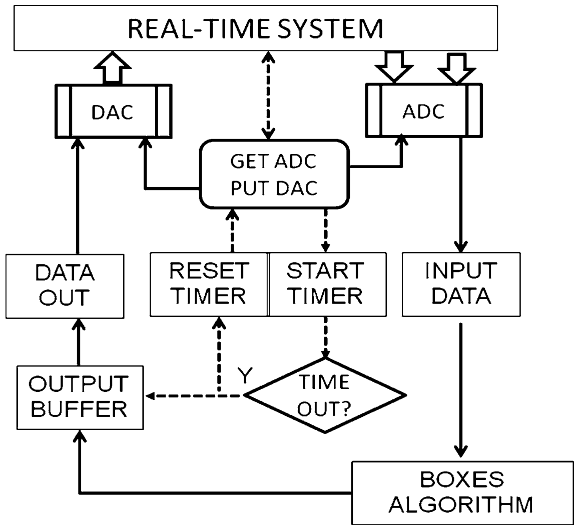


Fig. 4.7 Real-time timing loop

4.2.2 Continuous Systems that Sample Data

In a continuous application the reactive and reflective phases of the algorithm must execute within the same sample time. This dictates that the processing speed of the direct digital control algorithms can not include any lengthy procedures. In both the reactive mode and reflective modes, software speed is vital. Figure 4.7 illustrates the real-time loop that a continuous system might need to adopt.

What is important is that when the timer logic expires, a value must be sent to the output regardless of what state the BOXES algorithm is in. Missing perhaps an occasional control sample might be tolerable to the overall system dynamics but certainly not desirable.

In a continuous system other metrics are required. If the goal of the control game, for example, is to reduce oscillations in some state variable then the metric could be based on a measure of the sum of the areas between the first two oscillations of the response and the desired value. Figures 4.8 and 4.9 illustrate this.

Obviously, the sum of the values of the area under the curve ($A_1 + A_2$) in Fig. 4.8 is greater than that of ($A_3 + A_4$) in Fig. 4.9. This gives a meaningful discriminator of how well the system is performing or not, and can be used in a reward and penalty update scheme.

4.3 Update Procedures

As outlined in Chap. 3 the values in the signature table(s) that constitute control data are updated based on the overall performance of the system and how each cell has contributed to either an improvement or degradation of it. If the cell was

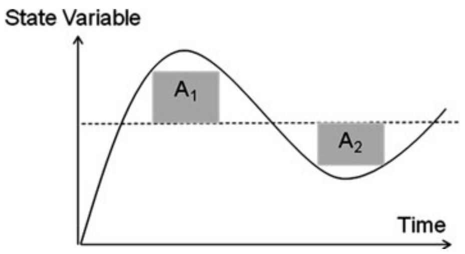


Fig. 4.8 Dynamic performance of system before learning

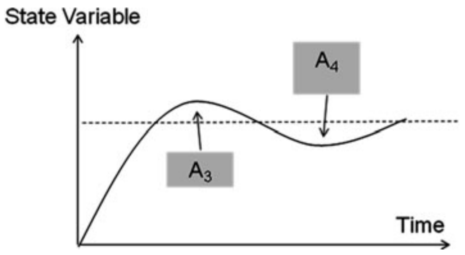


Fig. 4.9 Dynamic performance of system after learning

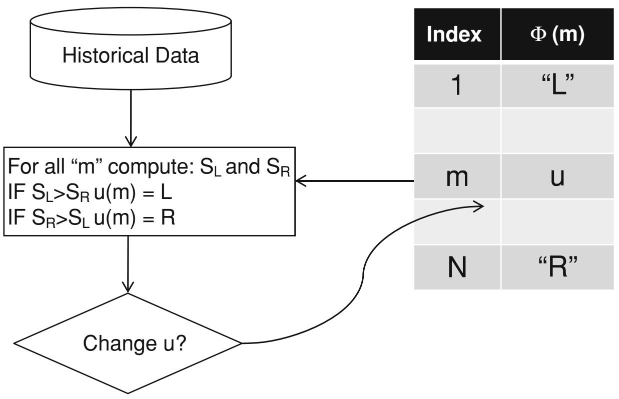


Fig. 4.10 Update signature table based on decision strength

entered many times, returning a control value of, say LEFT, and the system is not being controlled as expected, the update procedure must weaken the strength of that decision and eventually reverse it to RIGHT. Every box or cell uses its historical data and any immediately past data which is merged with the current performance metric of the system as a whole and some expectation criteria. This data changes over time consequently the control value alters frequently. However, it is to be hoped that after an initial training period most cells will arrive at a strong cell value. Figure 4.10 shows this for a system that controls a motor by providing *LEFT* and *RIGHT* decisions.

4.4 Concluding Comments

Having worked in the industrial manufacturing process control sector, the author is very well aware of the vast differences between actual, physical systems and simulations. In a simulation, practical real-time considerations are not as important because, for example, time is just another value on a printout or graph. Because the purpose of this book is to examine features of the BOXES algorithm much use is made of simulation, with the exception of Chaps. 6 and 7 which describe work on a real pole and cart system and that solved the auto restart problem using custom firmware.

References

1. Gustafsson T and Hansson J. Data management in real-time systems: a case of on-demand updates in vehicle control. Proc. RTAS 2004. 182–191.
2. Dorf RC and Bishop RH Modern Control Systems 12th edition (2008) Pearson Prentice Hall
3. Irving M. *Railway Systems Engineering in Action*. European Systems Engineering Conference 2006 18–20 September 2006 Edinburgh, UK
4. Durbin J and Koopman SJ Time series analysis by state space methods. 2001 Oxford University Press
5. Sjöberg J et al. Nonlinear black-box modeling in system identification: a unified overview **Automatica** Volume 31, Issue 12, December 1995, Pages 1691–1724
6. McGregor DR. Adaptive control of a dynamic system using genetic-based methods Intelligent Control, 1992, Proceedings of the 1992 IEEE International Symposium Page(s): 521–525
7. Russell, D.W. Failure Driven Learning in the Control of Ill-defined Continuous Systems *Int. J. of Cybernetics & Systems.* 1994: Vol. 25: Taylor & Francis. Washington DC: 555–566.

Chapter 5
Control of a Simulated Inverted Pendulum Using the BOXES Method

5.1 Introduction

The modeling and control of poorly defined, non-linear, real-time dynamic systems have always been classically problematic and mathematically approximate at best. An awareness of the faults of linearized models of real systems has caused researchers to turn to artificial intelligence (AI) for a possibly better source of solutions. The attractiveness of systems that utilize genetic algorithms, fuzzy logic, and neural networks, and their collected ability to tolerate uncertainty hardly needs to be mentioned. Perhaps the strongest feature that AI-based paradigms offer is their ability to adapt both parametrically and structurally.

The recognition and admiration of the rapidity and agility of the human problem-solving cognitive process has always been seen as a possible metaphor for the (optimal) solution of "wicked" [1] control problems that are difficult even for the usual AI-based systems to control because of unexpected unstable or chaotic behaviors that may occur during prolonged operation of the system. For example, as the components of a real-world system wear so the surfaces of controllability (presuming they exist) migrate nomadically inside the system domain. While initially not a problem, over time, a new problem may appear that is often temporally prohibitive or systemically impossible to control while the system is online.

One fairly well-known group of AI-based controllers utilizes a *signature-table* [2] mechanism in which decision intelligence is stored for rapid recall and periodically updated based on bespoke system criteria. The mathematical model of the dynamics of the system does not feature in the structure of the controller, which learns using a statistical inference procedure that is based on performance only.

While not perfect, a simulation of a system provides a useful platform on which to test control algorithms without construction costs and instrumentation. In this chapter, a well-known and accepted model of the trolley and pole system will be used for this purpose.

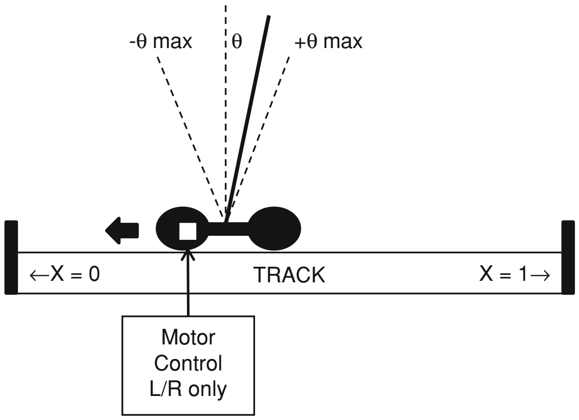


Fig. 5.1 The trolley and pole system overview

5.2 The Trolley and Pole Model

The pole and cart, or trolley and pole, is a familiar mechanically unstable dynamic system that is most useful in proving the efficacy of the BOXES control method. Stated simply, a motorized trolley is made to move in such a way as to keep a freely hinged pole swinging within some preset limits; the task is to keep the pole from falling while also keeping the trolley, which may skid under rapid acceleration and deceleration, on a finite track. This is not to be confused with the relatively simple swing-up pole-balancing systems that are readily available and a favorite in student laboratories.

The non-trivial solution to the pole and cart problem is to make the pole swing with steady bounded motion that is forced by moving the cart back and forth along a track. Figure 5.1 illustrates the system.

where,

a. State variables

x	Position of the trolley on a finite track
dx/dt	Velocity of the trolley on the track
θ	Angle of the freely swinging pole about the vertical
$d\theta/dt$	Velocity of the freely swinging pole

b. Parameters

u	Control value $\{+1$ or $-1\}$ from BOXES controller
F	Force on the trolley from a rapidly reversing motor
m	Mass of the pole
M	Mass of the trolley
a	Half the length of the pole
I	Moment of inertia of the pole
g	Acceleration due to gravity

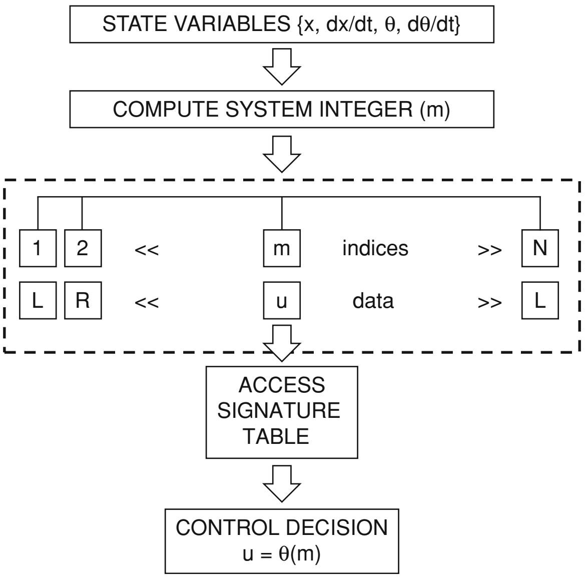

STATE VARIABLES {x, dx/dt, θ, dθ/dt}
COMPUTE SYSTEM INTEGER (m)
1
2
<<
m
indices
>>
N
L
R
<<
u
data
>>
L
ACCESS SIGNATURE TABLE
CONTROL DECISION u = θ(m)

Fig. 5.2 Signature table schema for the trolley and pole

The four state variables are related by Eqs. 5.1 and 5.2 which are shown in their nonlinearized form, but do ignore friction in the pole's hinge and slippage between the wheels and the track.

$$\frac{d^2\theta}{dt^2} = \frac{(m \cdot a \cdot g \cdot \sin(\theta) - m \cdot a \cdot \frac{d^2x}{dt^2} \cdot \cos(\theta))}{I + m \cdot a^2} \tag{5.1}$$

$$\frac{d^2x}{dt^2} = \frac{(u \cdot F - m \cdot a \cdot \frac{d^2\theta}{dt^2}\cos(\theta) + m \cdot a \cdot (\frac{d\theta}{dt})^2 \cdot \sin(\theta))}{(M + m)} \tag{5.2}$$

These equations will be used in the remainder of the book whenever a simulation of the trolley and pole is used. A solution of this model is fairly simple to obtain using numerical integration or a simulation package such as MatLab® and consequently provides a readily available system for use by control algorithms.

5.2.1 The Trolley and Pole Signature Table

The BOXES methodology [3] is an example of a signature-table controller Fig. 5.2 illustrates the particular schema for the trolley and pole where each table entry contains either an "L" or "R" motor direction.

Seminal work [4] on the control of a real inverted pendulum—in the form of a pole and cart—was performed in Liverpool and will be covered in more detail in Chap. 6. In this work both a simulation and physical rig were used. The problem is still a popular topic of study, for example [5]. The switching surface that an optimal solution lies along has been shown [6] to divide the state space into two clearly defined regions. A Pontriagin [7] analysis of the system model further confirms that optimal control is achievable using a *bang–bang* schema. This implies that an optimal solution is contained within a sequence of binary (LEFT or RIGHT) control decisions within the state space. The control task simply translates to providing the motorized cart with a correct sequence of LEFT and RIGHT switching commands.

The system is deemed to fail if any state variable falls outside of its own preset boundaries, in which case, the BOXES algorithm halts and authorizes an update of its signature-table control matrix and restarts the system.

5.2.2 Systems Engineering

To implement a BOXES-type controller for a simulation or model of the trolley and pole system, certain rules are necessary including:

- The motor on the trolley is to be digitally switched between full speed LEFT or full speed RIGHT only. In a simulation this is no problem as the value of u in Eq. 5.1 is simply inserted as ± 1 and the model instantly obeys any change in the direction of the motor's driving force. In reality, as will be shown further in the next chapter, this not at all easy because of the inertia of the cart, the whip action of the pole, spinning wheels, and the time lag inside the motor.
- The system will usually fail or need to be terminated by a timeout after a long run. In the simulation, this means the inclusion of some exit path that obeys a maximum simulated time for any run. This is usually at around 100 s.
- Learning is to be implemented only after the system fails or the run is aborted. This is so the system behaves in the *react* or *reflex* mode described above.
- The control matrix or signature table is populated initially with randomly selected LEFT and RIGHT decisions. In the simulation, this is not easy as the same set of *random* numbers will reappear if incorrectly seeded. One method of removing this bias is to use the system date and time as an unrepeatable seed for the computer's random number generator.
- Failure is defined by any state variable going out of range. This usually equates to the pole hitting a limiting value inside the cradle in which it is allowed to move in two dimensions, or the trolley hits an end-of-track marker.
- Reasonable initial conditions of the state variables are to be randomly generated. In the simulation this is not really a big problem, but a whole chapter (Chap. 7) is devoted to system restart in reality.

Online real-time control can be imposed on the simulated system as follows:

- Numerical integration of the state equations (Eqs. 5.1 and 5.2) produces the state variables. The simulated time is incremented by the integration interval in the model. A sampling strategy is enforced after some integer number of integrations and is a software configuration option.
- Computing the current state region identification integer, m, is based solely on the values of the state variables inside the integration interval.
- Indexing the signature-table control matrix, Φ, as an array obtains the current control value, $u = \Phi[m]$, where Φ exists as a set of +1 and -1 values representing the demand for LEFT or RIGHT motion and appears in the state equations (Eqs. 5.1 and 5.2).

5.2.3 An Overall Performance Metric

In order to evaluate the efficacy of control, it is commonplace to monitor the progression of *system merit* over many training runs. The basic merit of the system, for a trolley and pole, is an average value of controlled runtimes which are weighted by a system forgetfulness factor, δk. This factor downgrades the importance of early experience and eventually eliminates any inherent skewing produced by the random initial values implanted in the control matrix. Equations 5.3–5.5 show how merit is defined:

$$\text{global\_life} = \text{global\_life}^{*}\delta k + T_f \tag{5.3}$$

$$\text{global\_use} = \text{global\_use}^{*}\delta k + 1 \tag{5.4}$$

$$\text{system\_merit} = \frac{\text{global\_life}}{\text{global\_use}} \tag{5.5}$$

where

T_f	The duration of the last runtime before the system, failed
global_life	Aged average runtime over all runs so far
global_use	Aged experience value based on number of runs so far
δk	Forgetfulness factor, usually around 0.99

As the system learns to perform the task (of balancing the pole), the value of *system merit* increases and contributing cells should be rewarded. Conversely, if the trend is to produce a shorter than average performance runtime those cells that contributed to the decisions must be penalized.

5.2.4 The Importance of State Boundaries

It is obvious that the location and number of boundaries for each of the state variables attached to the BOXES system is of paramount importance. Russell [8] in a critique of the BOXES method observes that these boundary values present a "potential flaw in the methodology." However, within this "flaw" may lay a novel mechanism for self-organization which will be shown in Chap. 13.

To simplify the process, all *N* state variables are normalized before entering the state allocation algorithm. It is assumed here that a priori knowledge of the maximum and minimum value for each variable is available. Equation (5.6) shows the process that transforms a state variable $var_i\ (t)$ into V_i the *i*th normalized value.

$$V_i = (V\max_i - var_i(t))/(V\max_i - V\min_i) \tag{5.6}$$

where,

$var_i\ (t)$	Current sampled value of the ith state variable
$V\max_i$	Maximum value of the ith state variable
$V\min_i$	Minimum value of the ith state variable

If each of the normalized state variables (*Vi*) is sub-dividable into N_{bi} zones then it is a simple algorithmic process to assign individual zone numbers to each of them.

5.2.5 Computation of a Unique System Integer

In the BOXES method, the set of zone integers $\{i_1 \ldots i_{n\}}$ defines the system at any time within a given sample period. If the number of boundaries for each variable is contained in the set $\{Nb_1 \ldots Nb_n\}$, simple integer arithmetic yields a unique state number *m* for any given set of zones as seen in Eq. 5.7.

$$m = i_1 + \sum\left(\prod_{i=2}^{N} \cdot \prod_{k=2}^{i-1} \cdot (Nb_i) \cdot (i_k - 1)\right) \tag{5.7}$$

In the classic work by Michie and Chambers [3], for a trolley and pole system, the state variables *x* (the trolley position) and θ (the pole angle) were divided into five regions (i.e. $Nb_1 = Nb_3 = 5$), whereas dx/dt (the trolley velocity) and dθ/d*t* (the pole velocity) were divided into three (i.e. $Nb_2 = Nb_4 = 3$). Applying end values {1, 1, 1, 1} and {5 ,3, 5, 3} for each zone, Eq. 5.7 for the four state variables ($N = 4$), yields Eqs. 5.8 and 5.9:

$$m\{1,1,1,1\}, = 1 + (5*0 + 5*3*0 + \ 5*3*5*0) \ = \ 1 \tag{5.8}$$

$$m\{5,3,5,3\} = 5 + (5*2 + 5*3*4 + 5*3*5*2) \ = 225 \tag{5.9}$$

This means that the control matrix needs to have 225 elements (5*3*5*3), which was indeed the case. If the value of any variable strays outside of the posted minimum or maximum values, the system is deemed to have failed, and a zero value of *m*, the system integer, returned. For every non-zero value of the system integer, there exists a signature-table data point that represents some LEFT/RIGHT control directive.

5.3 Simulation Software

The BOXES simulation software can be divided into two phases, although later another phase (evolution) will be added in Chap. 13. Because the BOXES algorithm uses what can be many training runs before it establishes good control behaviors it is important to carefully design campaigns of runs. Inside any campaign, individual run simulations are performed which because of the large number of iterations, are kept largely invisible to the user. The individual run section is the second phase of the simulation and must be designed very carefully to avoid numerical traps and artificial local minima, such as when the pole might jitter about some boundary point that could latently skew the learning process.

5.3.1 The Campaign Set Up Phase

The first phase deals with the assignment of physical values for the pole and cart, its motor power, and ranges for each of the state variables as they appear in Eqs. 5.1, 5.2, 5.6, and 5.7. Beyond this, the learning parameters and individual state boundary values are set for the duration of the campaign. Additionally, random values are assigned to the signature-table data values (LEFT/RIGHT) in this phase. Because of the large number of runs in a campaign, which is sometimes upwards of several hundreds of thousands, a scheme for archiving data at selected intervals is essential because the user cannot inspect every instance. Figure 5.3 illustrates this important phase in the simulation process. Having a careful design of experiments approach prevents the loss of much time in evaluating long computational campaigns in retrospect.

Data generated during a campaign of say 100,000 runs can now be extracted from the campaign log files and analyzed such as given in Fig. 5.4.

5.3.2 The Individual Run Phase

During each run, the software accesses any fixed run data and generates the state variable function by numerical integration. Of course, the independent variable,

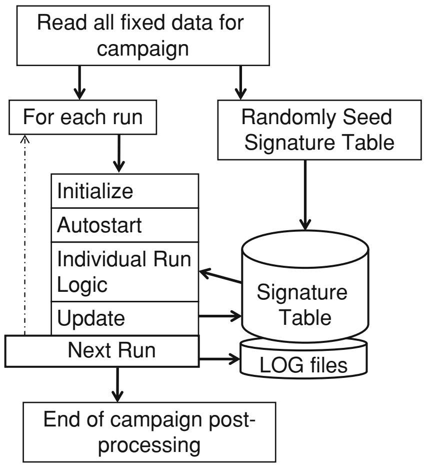

Read all fixed data for campaign
For each run
Randomly Seed Signature Table
Initialize
Autostart
Individual Run Logic
Update
Next Run
Signature Table
LOG files
End of campaign post-processing

Fig. 5.3 Simulation campaign set up

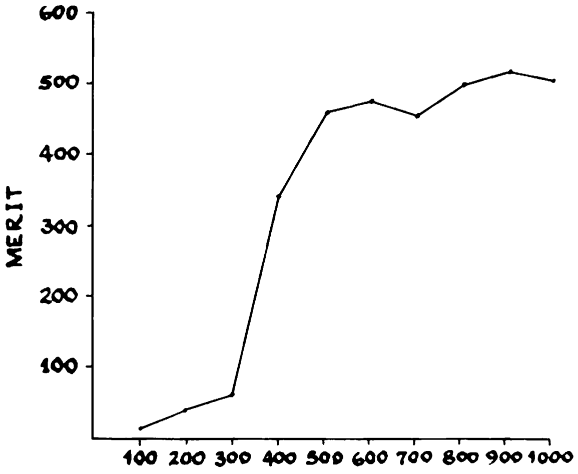
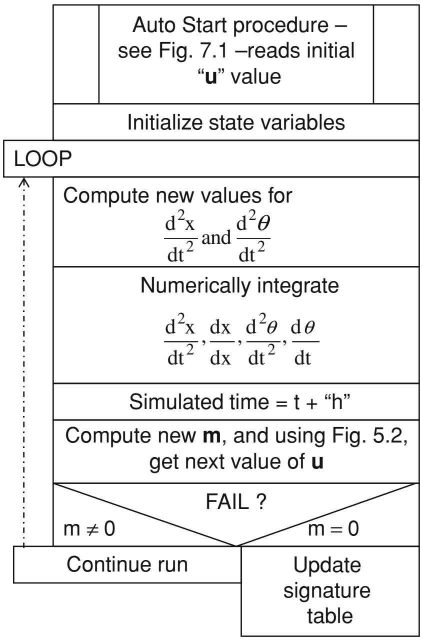


Fig. 5.4 Results showing merit increases over 1,000 learning runs

time is incremented after each integration step. Provided some mathematically oriented compiler is used, it is not really that important which operating system or language is used. In Chap. 6 when the controller is linked to a real system via custom firmware this may not be so.

Figure 5.5 shows the inner phase of the simulation in which the trolley and pole are randomly started, allowed to access the current values in the signature table, until the system eventually fails, or reaches some preset plateau of performance. In Fig. 5.5 the equations are generated from Eqs. 5.1 and 5.2 and only need access to the single valued data contained in the signature table. At the end of each run, the statistical data behind each of the BOXES cells in the signature table are updated even if they did not contribute to the last run or will not be accessed possibly for several runs in the future. The log file can include records that reflect



Auto Start procedure – see Fig. 7.1 –reads initial "u" value
Initialize state variables
LOOP
Compute new values for $\frac{d^2x}{dt^2}$ and $\frac{d^2\theta}{dt^2}$
Numerically integrate $\frac{d^2x}{dt^2}, \frac{dx}{dx}, \frac{d^2\theta}{dt^2}, \frac{d\theta}{dt}$
Simulated time = t + "h"
Compute new m, and using Fig. 5.2, get next value of u
FAIL ?
m ≠ 0
m = 0
Continue run
Update signature table

Fig. 5.5 Outline of individual run simulation software

how often any of the 225 LEFT/RIGHT motor direction decisions reverse during a campaign.

5.4 Simulation Results

As the system functions, data is collected by the BOXES algorithm. This may be a count of the number of times any particular state (or "box") is entered, or a summation of the amount of time that the system remained in repeated entries into any given state. Because the task is to balance a pole for extended periods of time, it is not surprising that the time the system dwells within any given cell is important. Data is accumulated for each entered state only, which implies that all states are not going to be entered during any specific run, and as such are non-contributors to the control this time. If the initial conditions of all variables are always set to be random, yet rational, values during the autostart process, the system will be forced into exploring most of its complete operational domain.

Some startup conditions occur that present an impossible situation to the system, yet even they can be useful. For example, if the trolley is started near an end of its track with the pole facing outward and the motor direction is such that it almost immediately drives the trolley into a state of failure, the short time-to-fail value (Tf) will cause the system merit to be reduced, and the strength of the decision in the offending cell will be reduced. While, hopefully, not too many instances of such failures occur, eventually BOXES will reverse that decision giving the system time to pull the trolley and pole away from that situation.

Of course, if the trolley is moving too rapidly near the ends of the track, which the autostart function may generate by chance, then the system reaches a point of no return regardless of the motor direction. In Chap. 7 an autostart sub-system is described that attempts to start the simulations with state variables that are both random yet reasonable in that the system is at least given a chance of recovery and learning. This is akin to the familiar chess player in Chap. 2 starting a game with the board in a hopeless position. If a novitiate of the game were only presented with situations such as this, the opportunity for advancement and learning would be very limited.

5.4.1 Typical Results

Figure 5.6 shows a graph of merit versus number of runs generated from a typical campaign using the original BOXES algorithm applied to a trolley and pole system simulation.

It is apparent from this figure that learning is neither regular nor monotonic, and disappointing groups of runs do occur during a campaign. This is not uncommon in the BOXES genre of learning systems. The reason for poor performance could be that the system was exploring naïve regions of the signature table or just recovering from unreasonable initial data decision values.

5.5 Update of Statistical Databases

At the end of every learning run, the system merit is recomputed using Eq. 5.3 through 5.5 and reflects the current overall performance or skill level of the system. To effect learning, the BOXES post-processing algorithm must now update the statistical database for each cell by merging the data accumulated during the previous run with statistics stored from all previous runs. Because the decision context is binary (e.g. LEFT/RIGHT) two sets of statistics must be kept for each value of m, one for either value.

If the heuristic uses a count of entries into the boxes, the data elements that would need recording for the ith cell might be called $left\_count_i$ and $right\_count_i$. If the time spent inside that same cell was selected as another statistical parameter, the data elements would be called $left\_life_i$ and $right\_life_i$. If the prior run had used the ith cell n_i times and had spent tim_i (time units) in that cell, and if it was currently supplying LEFT as its knowledge element, Eqs. 5.10–5.13 show how the statistics would be updated according to the BOXES method.

$$\mathrm{left\_count}_i = \mathrm{left\_count}_i * \delta k + n_i \tag{5.10}$$

$$\text{right\_count}_i = \text{right\_count}_i * \delta k \tag{5.11}$$

$$\text{left\_life}_i = \text{left\_life}_i * \delta k + tim_i \tag{5.12}$$

$$\text{right\_life}_i = \text{right\_life}_i * \delta k \tag{5.13}$$

In the case of the trolley and pole system, sometimes the *time to fail* (Tf) is used instead of the *time spent in each cell* during the last run (tim_i). Aging both phyla is appropriate in order that past data is uniformly rated as being less and less significant.

5.5.1 Determination of Decision Strength

Once the statistical database has been updated to include the current run, and a new overall system merit identified, the BOXES algorithm assigns statistical strengths for both control decisions. Equations (5.14–5.16) are typical of this calculation, again shown for the *i*th cell.

$$\text{dla} = C_0 + C_1 * merit \tag{5.14}$$

$$\text{left\_strength}_i = \frac{(\text{left\_life}_i + K * \text{dla})}{(\text{left\_count}_i + K)} \tag{5.15}$$

$$\text{right\_strength}_i = \frac{(\text{right\_life}_\text{i} + \text{K} * \text{dla})}{(\text{right\_count}_\text{i} + K)} \tag{5.16}$$

where,

C_o	Learning constant coefficient
C_1	Learning rate coefficient
dla	Desired level of achievement
K	Desired learning rate

In the learning phase, the decision matrix, $\Phi[i]$, for each cell is updated based on the value of the statistical strengths of each decision using Eqs. 5.17 and 5.18.

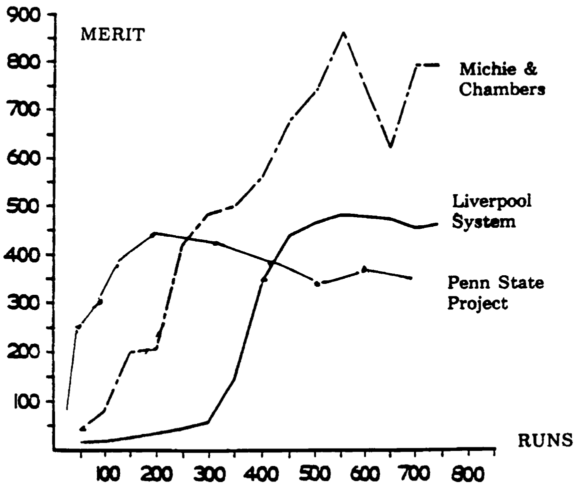
$$\text{IF left\_strength(i)} > \text{right\_strength(i) THEN } \Phi[\text{i}] = \text{``LEFT''} \tag{5.17}$$

$$\text{IF right\_strength(i)} > \text{left\_strength(i) THEN } \Phi[\text{i}] = \text{``RIGHT''} \tag{5.18}$$

If by chance the two strength values are equal, the standing decision is left unchanged or re-randomized.

Several versions of the simulation exist and Fig. 5.6 shows a comparison between results from three versions of the software. As expected the actual

Fig. 5.6 Comparison of results by various authors



numerical values vary from project to project, but all three show a similar, quite rapid improvement, which is the objective of the figure. It should be noted that the performance factor (merit) does not increase monotonically. As each run starts from a different region in the solution space, some areas are not as mature and therefore less certain of their decisions, and may consequently return motor control directions that cause early failure.

5.5.2 Near-Neighbor Advisor Cells

Because the ideal location of boundaries between state integer values is unknown, it may be possible to use data from near neighbor advisor system integers in decision making. In subsequent studies [9], the author discovered that the performance of the learning algorithm could be much improved by the addition of advisor logic into the decision matrix update procedure. There are several advisor schemas including winner-takes-all, voting and statistical strength aggregation. These will be discussed in more detail in Chaps. 8, 9, and 10.

In both cases, it is desired that the advice from these selected cells effectively reinforce the decision stored at the current state integer index or that they provide a compelling reason to reverse the decision. To avoid the possibility of a *blind leading the blind* scenario, any advisors that offered advice during a run are rewarded and punished during the update phase. While this added recursion in the software does increase the associated time lag of the computation, it provides what amounts to 360° feedback to the advisors.

5.6 Conclusions

This chapter has indicated how simulation software can be written to produce campaigns of control runs that use the BOXES algorithm as the primary control method. If the algorithms illustrated above are implemented with a good numerical integration routine, the overall system merit will increase over time showing that it is successfully learning the game of pole-balancing. The primary learning intervals are punctuated by anticipated *failure* events. A usual failure condition is that the pole angle exceeds a recoverable value, or the trolley reaches the end of its track. In the simulation situation, for a pole and cart, it is almost inevitable that the pole and cart model will fail. If not, and the system is under control for more than some maximum time threshold, an end-of-run state is asserted by the algorithm.

In real dynamic systems, the unavoidable end-game incident may also be a timeout or programed interrupt, a component failure, or a critical communications fault.

References

1. Blum, B.I. 1996. *Beyond Programming: to a New Era of Design,* 132–134. New York: Oxford University Press
2. Canudas de Wit, C. A. 1988. *Adaptive Control for Partially-known Systems*, 99ff. Amsterdam: Elsevier
3. Michie, D. and Chambers, R.A. 1968. BOXES–an Experiment in Adaptive Control. *Machine Intelligence II*, 137–152. London: Oliver & Boyd
4. Russell, D.W., Rees S.J. and Boyes, J.A. 1977. A Micro-system for Control by Automata in Real-Life Situations. *Proc. Conf. on Information Sciences and Systems.* Johns Hopkins University, Baltimore. 226–230
5. Widjaja M. and Yurkovich, S. 1995 Intelligent Control for Swing Up and Balancing of an Inverted Pendulum System. *Proc. 4th. IEEE Conference on Control Applications*, Albany New York. 534–542.
6. Russell, D.W. and Rees, S.J. 1975. System Control—a Case Study of a Statistical Learning Automaton. *Progress in Cybernetics Systems Research* 2:114–120, New York: Hemisphere Publishing Co.
7. Rees, S.J. PhD Thesis. Liverpool Polytechnic, 1977
8. Russell, D. W., 1993. A Critical Assessment of the BOXES Paradigm. *J. Applied Artificial Intelligence* 7(4): 383–395
9. Russell, D.W., 1995. Advisor Logic. *Control Engineering Practice* 3(7): 977–984 July, Oxford: Pergammon Press.

Chapter 6
The Liverpool Experiments

6.1 Introduction to Reality

While the simulation studies proved useful in algorithmic development, it is only a real mechatronic system that can prove out the method and allow comparisons with a human operator. Humans are amazingly facile in learning tasks such as playing chess, driving a car, or riding a bicycle yet suffer from inherent weaknesses such as a lack of long-term concentration, susceptibility to panic in an emergency situation, and general irrationality in decision making. This chapter shows that the BOXES algorithm, implemented in a mechatronic system has none of these human traits and is not distracted from its task of balancing the pole by moving a trolley once a decision is made. In short, it can outperform even trained human operators.

In order to illustrate this, the Liverpool rig [1] described below includes a feature that allows a human operator identical access to the system's control unit as the BOXES automaton. Because the BOXES methodology control mechanism uses values from an initially random signature table, it is expected that repeated training runs will be required before any vestige of learned prowess becomes evident. It was therefore expected, and only fair, that human operators would also need some training experience to increase their proficiency at the balancing task.

6.2 The Liverpool Trolley and Pole Rig

To control a real-world trolley and pole system using the BOXES method requires the construction of several computer and hardware [2] interfaces. Figure 6.1 illustrates the overall system design of the Liverpool rigs.

Written with John Boyes, Liverpool University, retired.

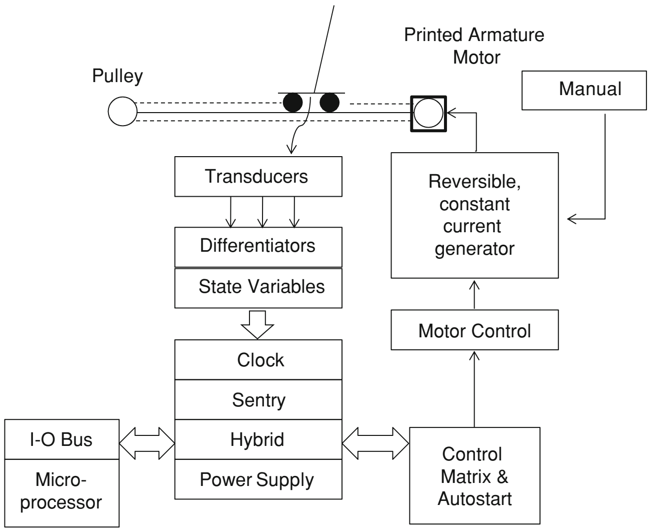


Fig. 6.1 Overall system design

As will become more evident in this chapter there were a host of unforeseen situations that arose when dealing with the control of this real dynamically unstable system.

6.2.1 Practical Aspects of the Liverpool System

A Fletcher's apparatus [3] was used because of its inherent property of low friction. The first trolley that was built was rather light and narrow and very unstable in the transverse direction so it was re-designed using heavy aluminum. Fortunately, the track was wide enough to allow for this improvement. The pole was mounted in the center of the trolley on a ball race and limiters fitted so that the pole was unable to fall any further than 20° on either side of center; this being well outside of the working range. Micro-switches were fitted to the limiters for use in the "sentry" logic which ensured that, even if fail states went undetected, the experiment could be curtailed before any damage was done to the apparatus (or the operator). Micro-switches were also fitted to the ends of the track in order to detect uncontrolled overruns and enable the sentry system to shut down the motor power.

The drive was transmitted from the motor to the trolley via a length of cord that contained a fiberglass core in order to minimize stretching. A 4′ (100 mm) diameter pulley wheel was used which produced a force of 4.6 N to be applied to the trolley when the motor current was set to its maximum value. The magnitude of the currents was adjustable using variable resistors that were set manually. A spring balance was attached to the trolley and the current was turned on and adjusted until the required force was obtained. This was then repeated to obtain an equal force in the opposite direction.

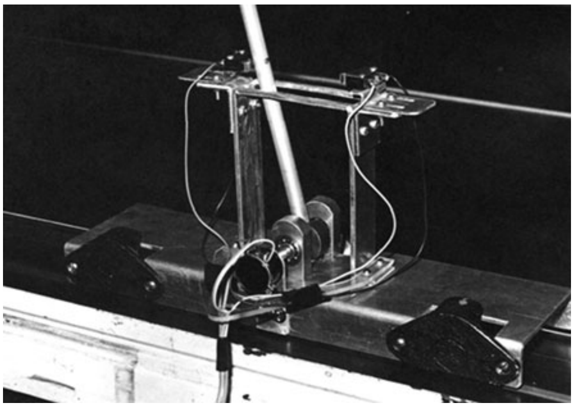


Fig. 6.2 The Liverpool trolley

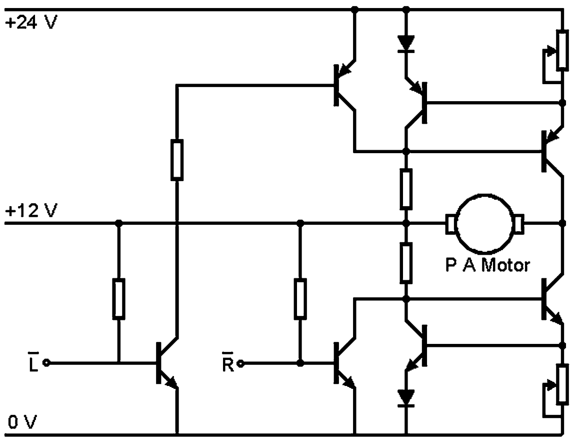


Fig. 6.3 Schematic of the constant-current generator

A manual controller was constructed in a small aluminum box into which four switches were mounted; three on the top row (left, auto-start, and right) and a fourth switch with a wider button below these that served as a panic stop function. In the original rig the force applied to drive the trolley was to be of fixed magnitude and allowed to vary only in the direction of its motor. To obtain this fixed force, a printed armature or pancake motor [4] was chosen to provide a ripple-free driving torque. This type of motor is characterized by the lack of iron in its construction and thus the absence of any associated non-linear losses. The torque produced by this type of motor is directly proportional to the applied current. Figure 6.2 shows a photograph of the trolley system.

To ensure smooth performance it was necessary to design and construct a constant-current controller. In order to cope with the currents required, power transistors were used and, to facilitate the reversing function, a push–pull arrangement of two constant-current generators was implemented. For reference, Fig. 6.3 shows the circuit diagram of the constant-current generator.

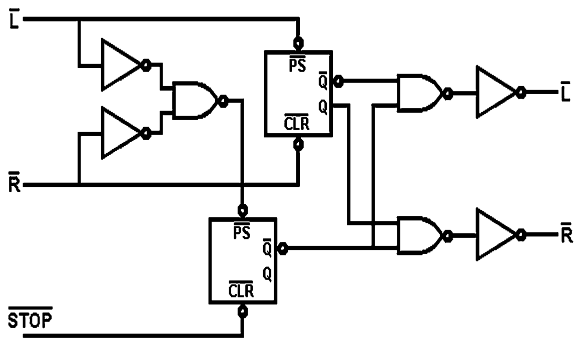


Fig. 6.4 Schematic of a tri-state constant-current generator

The circuit was designed to meet the requirement that the system be fail-safe so that if there was a loss of control signal, for example, due to a cable becoming disconnected, the system would shut down. Although the circuit was locally fail-safe in the event of disconnection, an operator could still press both left and right buttons or the computer program could detect an out of control situation and do the same. For this reason the circuit shown in Fig. 6.4 was developed to ensure that only one of the three motor conditions; left, right, and stop could ever be selected at any one time.

6.2.2 Instrumentation and the State Variables

For the instrumentation of the rig variable resistors were used to obtain the position of the trolley (x) and the angular position of the pole (θ). The x transducer was a multi-turn potentiometer that doubled as the idler wheel at the opposite end of the track to the motor. Both transducers were energized with ± 15 V so that the central positions gave outputs of zero volts. In order to obtain the velocities, differentiators were used. However, electronic differentiators can be inherently extremely unstable due to the very high gain presented to noise and other high frequency signals. To overcome this, a band pass filter was included so that the circuit would act as a differentiator within the operating range, which is for signals with frequencies up to a few hundred hertz and act as an integrator for frequencies above that value. This meant that the low frequency signals from the transducers could be differentiated without the higher frequencies being allowed to introduce instability.

The next stage in the signal processing might seem to warrant the use of analog-to-digital converters but to reduce processing in the micro-computer a simpler solution was sought. The following describes the function of the state allocator system that was designed to perform this function. The state integer inputs to the electronic state allocator (known as the M-Box) were designated I, J, K, and L corresponding to the state variables x, dx/dt, θ, and dθ/dt, respectively. The output signal which was to be the overall system integer, representing the combined state, was given the usual designation of m. The concept behind this design

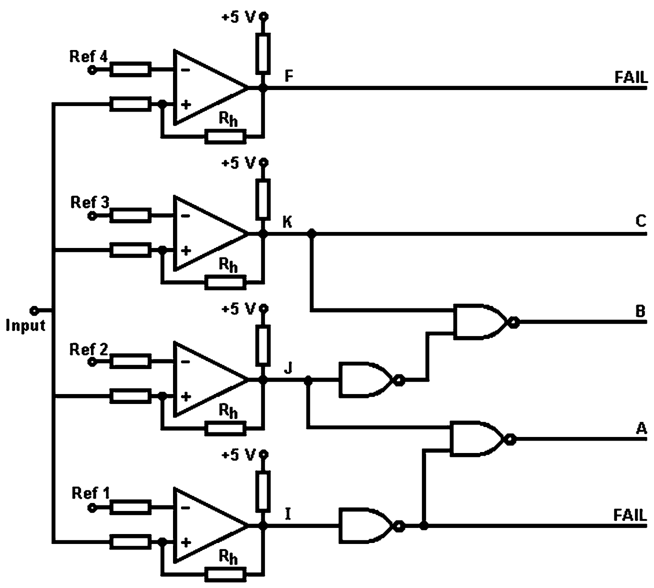


Fig. 6.5 Schematic of a three value state allocator

was to rapidly produce an integer number (m) that would represent the system state directly from individual outputs of the transducers and their differentiators.

The following is an example of converting the velocity ($\mathrm{d}x/\mathrm{d}t$) into its relevant state number which throughout has been a number in the range of 1–3. A set of four comparators was used with some simple logic to encode the signal instantaneously, on command, to one of those three values (or a FAIL zero state) using the circuit shown in schematic form as Fig. 6.5.

In this case, the signals at points A, B, and C are asserted only when the equivalent state is entered. Similar circuits were designed for $\mathrm{d}\theta/\mathrm{d}t$, x, θ but of course, in the latter two cases, six comparators had to be used because the ranges of x and θ were divided into five states and not three.

For the comparator reference voltages, precision variable resistors with calibrated control knobs were fitted to the front panel of the console so that the state boundaries could be easily modified without having to re-program the computer.

At this point it was realized that the number of variable resistors could be reduced if the state boundaries were symmetrically spaced on either side of zero. This was a reasonable assumption in the early designs of the rig although not part of the final design. The analog outputs from the transducers were sent via precision rectifiers to a modified state allocator together with another signal to indicate whether movement was to the right (positive) or to the left (negative). This system not only reduced the amount of hardware (and cost) but also made it much simpler to make adjustments to the state boundaries.

6.2.3 Manual Auto-start

In the case of manual control, the handover from auto-start to human control was very simple; the auto-start simply abandoned the task and left it up to the human controller to take over at the appropriate time, for example, after four direction reversals. In practice, the human controller could press the left and right buttons at any time but they were simply ignored until the auto-start sequence had completed. When the computer was used to control the system, the program would go into a waiting loop looking at a flag bit on the interface board. When the auto-start completed, it would assert the flag bit and the computer would take over control of the trolley. More about the auto-start feature is contained in Chap. 7.

6.2.4 Driving the Trolley

It was stated earlier that every effort was made to alleviate friction so that the real system would emulate the model more closely. This turned out to be far more difficult than it first appeared. The problem was due to the tension that was required in the lacing cord; it had to be tight in order to avoid slippage and thus produced a loss of datum on the x transducer but, the tighter it was, the worse the bearing friction became. While both the printed armature motor and the multi-turn potentiometer had very low rotational friction, applying radial forces due to the tension of the control cord caused considerably more friction than was expected.

The upshot of this became clear when an experiment was carried out to test the drive system. In this test the rig was placed in a darkened room and a camera, set to a very long exposure, was triggered. A stroboscope was started and the constant-current generator turned on. The pole images were measured on a traveling microscope and a difference table constructed from the results. On analysis, it turned out that the applied force was indeed constant but not as high as was calculated when assuming zero friction. Thus when accelerating the effective force, which should have been 1 N was in fact only 0.58 N and this meant that the effective decelerating force was 1.42 N. It turned out, however, that the difference in accelerating and decelerating forces made very little difference to the operation of the rig and meaningful results were still readily obtained from it. One major benefit of this design was the fact that, on command from the algorithm, the motor direction reversal was virtually instantaneous.

6.2.5 The Microprocessor

Data from the physical system passes along the I-O bus and is presented to the BOXES program at some fixed sampling rate. This required the creation of an I-O driver that ensured timely values of the state integer became available to the BOXES logic. In future experiments it is anticipated that the algorithm will execute on an on-board microprocessor with local data collection and control functionality. In the original Liverpool rig many functions were implemented in assembler language on an available minicomputer.

6.2.6 The BOXES Algorithm and the Real-Time Monitor

Using the system outlined in Fig. 6.1, the same BOXES algorithm executed on the microprocessor as that used in simulations. The real-time operating system monitor was different as it had to read data and return motor control signals along the I-O bus rather than simply storing them in software structures as part of a polling scheme. The software system was interrupt driven and largely custom built.

6.3 Systems Engineering

To implement a BOXES-type controller for a real trolley and pole system, it is a requirement that the motor on the trolley be digitally switched between full speed ahead LEFT and full speed ahead RIGHT only. In a simulation this is no problem as the value, given as u in Eq. 5.1, is simply asserted as plus or minus unity and the model instantly reacts to a request for a direction change. In reality, as will be shown in the next chapter, this not at all easy. From a software systems view the following points are salient:

- The system will usually fail or need to be terminated by a timeout after a long run. In the simulation, this means the inclusion of some maximum simulated time, usually around 100 s for any run. In the real system, failure occurs after a much shorter time interval. It is the park and restart sequence that is difficult.
- Learning is to be implemented only after the system fails or the run is aborted in order for the system to behave in the REACT/REFLECT mode.
- The initial control matrix must be randomly seeded with LEFT and RIGHT decisions. In the simulation and the real system, this is not easy as the same set of "random" numbers appear if incorrectly seeded. One method of removing bias is the use the system date and time as an unrepeatable seed for the calculation. The firmware array is preloaded from a minicomputer in the real system while the software array is loaded in the simulation.
- Failure is defined by any state variable going out of range. This usually equates to the pole hitting a limiter, which is a cradle inside which it is allowed to move but only in two dimensions, or the trolley hits an end-of-track stopper.

Online real-time control can be imposed on the system provided the following rules and conditions are applied:

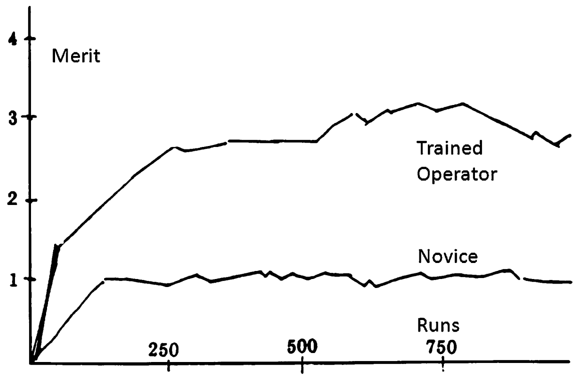


Fig. 6.6 Merit results for trained versus untrained operators

- Consistent real-time signal processing of state variables.
- Computing the current state region identification integer, m, based solely on the sampled values of the state variables.
- Indexing the signature table control matrix, Φ, to obtain a control value, $u = \Phi[m]u = (\Phi[m]$, where Φ exists only as a set of +1 and -1 values, representing the demand for LEFT or RIGHT motion.
- Returning u as the closed-loop control decision to the system.
- Enforcing the decision using a rapid reversal motor (e.g. printed armature).
- Sampling the state variables at a sufficiently rapid rate.

6.3.1 How Boundaries on Each State Variable were Imposed

It is obvious that the location and number of boundaries for each of the state variables attached to the BOXES system is of paramount importance. Russell [5] in a critique of the BOXES method observed that these boundary values may present a potential flaw in the methodology. It is the thesis of this monograph that within this flaw may lay a novel mechanism for self-organization as will be illustrated in Chap. 13.

To simplify the process, all of the state variables are normalized before entering the state allocation algorithm. It is assumed here that knowledge of the maximum and minimum values for each variable is available.

6.4 Results from the Liverpool Rig

Figure 6.6 shows actual results from the rig comparing values of merit awarded to a novice compared to a trained human operator after about 1000 training runs.

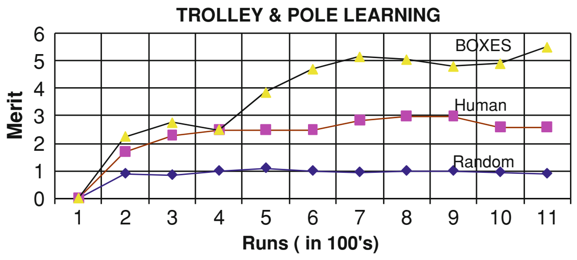


Fig. 6.7 BOXES versus human and random operators

The trained human hardly ever managed to balance the system for more than 10 s and as the merit figure shows actually averaged closer to 3 s. In Chap. 5, simulation studies yielded merit values of over a 100 s fairly quickly, so there is an obviously disjoint between simulations and the real world. This is a well-known fact which can cause the performance of a real system to be disguised by the very models that are built to facilitate design features. It goes without saying that models are useful in the preliminary design phase of a system but certainly not necessarily a definitive predictor of real-time performance.

Figure 6.7 shows a comparison of merit achievement of the BOXES algorithm versus a skilled operator and a random signal generator.

6.5 Conclusions

What are most obvious in this chapter are the smaller values of merit that are achieved in the physical rig as compared to the simulation, and the need for custom built firmware in order to get the system to work. This is understandable considering the unknowable factors such as wheel slippage, unavoidable time delays within the motor, and the like. The simulation operates in an almost perfect world, whereas the real trolley and pole system does not. What was most encouraging was the ability of the BOXES algorithm to learn to outperform a human being in the inverted pendulum task.

In these experiments it was determined that the construction of some custom hardware and firmware was acceptable so that the algorithm could be tested without the introduction of any other operating system or connectivity complications.

Looking more generally, it is always a matter of conjecture that an automaton can be given complete autonomy in a control task. Grant [6] succinctly states that three scenarios exist. The first is where the BOXES algorithm just accepts the decisions from the human, without effecting any decisions itself. The second allows no provision for the human to give a decision rather deferring to the algorithm, so that the decision making is autonomous. In the third case, some criterion may govern whether the algorithm has enough confidence in its decision to override any decision that the human might take.

In any dynamic system controlled by a learning algorithm, the unavoidable end-game incident may be a timeout or programed interrupt, or component failure which may cause a catastrophic or life-threatening situation to occur. It is because of this that systems such as the BOXES method are best suited to enhancing existing failure-proof controllers as will be shown later in the text.

References

1. Russell, D.W., Rees S.J. and Boyes, J.A. 1977. A Micro-system for Control by Automata in Real-Life Situations. *Proc. Conf. on Information Sciences and Systems.* Johns Hopkins University, Baltimore. 226–230
2. Boyes J. 1977 A Man-Machine Interface for Training an Automaton for a Real-world Situation. MPhil Thesis; Liverpool Polytechnic
3. Fletcher's Trolley Lab http://www.freewebs.com/dazed42/phys.htm
4. Printed Motor Works. http://www.printedmotorworks.com/printed-motors/
5. Russell DW, 1993. A Critical Assessment of the BOXES Paradigm. *J. Applied Artificial Intelligence* 7(4): 383–395
6. Grant, S 1990 Modelling Cognitive Aspects of Complex Control Tasks. PhD Thesis; University of Strathclyde Department of Computer Science

Chapter 7
Solving the Autostart Dilemma

7.1 Introduction to the Autostart Dilemma

The BOXES methodology strives to achieve as true a black-box mode of control as is possible. This means that there should always be a minimum amount of a priori system information needed to implement the system. In previous chapters it was stated that every system needed values for the minimum and maximum values of each state variable so normalization can be achieved. Other required system knowledge was the individual distributions of zone boundaries for each normalized state variable. Both of these do not contribute to the dynamic behavior of the system. With the goal of keeping the amount of specific system data to a minimum, it is clear that how the system starts up therefore cannot follow some preset favorable pattern; rather, while being reasonable, it must be as random as possible. This chapter focuses on how this is accomplished. There are two cases to be considered; each one causes the simulation or real system to begin its dynamic motion in a random manner, while ensuring that in both situations all starting values are valid and connected to the signature table.

Section 7.2 describes how a simulated system can be restarted using software and a random number generator. Section 7.3 uses a system catch up method that can be applied to simulations or to a physical system. Section 7.5 briefly shows how the system can be manually restarted. Figure 7.1 explains these options schematically. Regardless of how the system start up is effected, the mechatronics of the system must not allow data collected in this preliminary phase to corrupt the real run data when the system is under BOXES control.

7.2 Random Restart Simulation Software

When the trolley and pole simulation is to be run repeatedly so that its learning capability can be recorded, a random autostart procedure becomes essential. A typical simulated run campaign might entail a million such runs. In this section, an

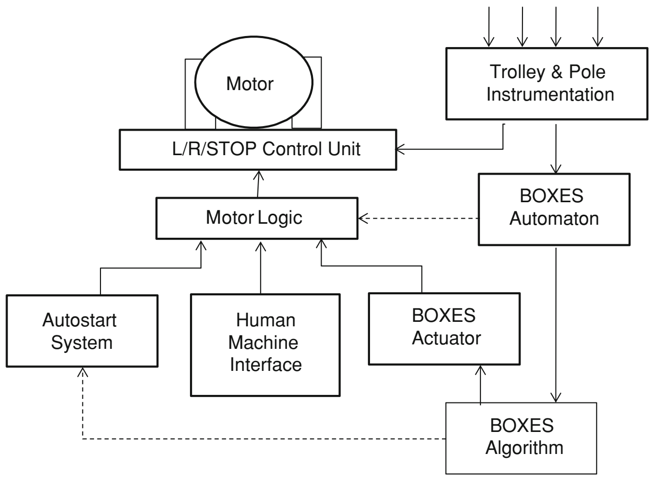


Fig. 7.1 Start up mechatronics

algorithm is described that places each state variable randomly within its range of legal values and using those values reads an initial motor control value from the current signature table before starting the actual simulation.

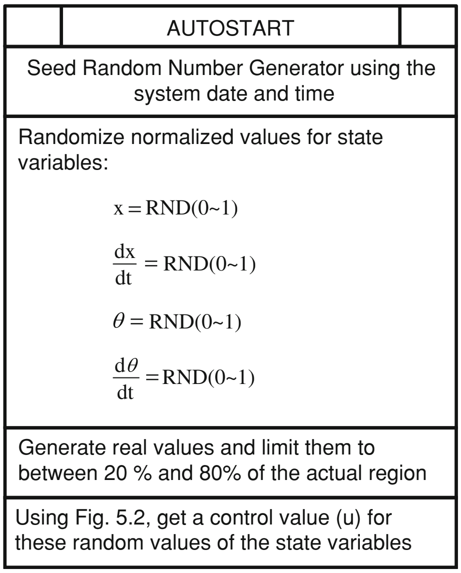
7.2.1 Restart Software

Section 5.2.6 described a simplified version of the overall simulation software design and the autostart module was referenced in Fig. 5.3. Figure 7.2 is an expansion of that logic for a simulated system.

In a campaign of simulations, it is essential to avoid accessing the same sequence of random numbers. In most compilers it is possible to seed the random number generator with a starting value from which all random numbers are produced. Using the same seed value is a good feature for reproducibility, but inside a long series of simulations, where true randomness is being sought in every successive restart, it is advantageous to use the system date and time as a unique non-repeating seed value.

7.2.2 Random Initial State Integers

Table 7.1 shows the first ten data values for the state variables generated from a BOXES simulation using the random state integer method of initialization. In this data set, the value of x (the trolley position) varied between ±1.2 m, dx/dt (the velocity of the trolley) varied between ±0.45 m/s, θ (the pole angle)


AUTOSTART

Seed Random Number Generator using the system date and time

Randomize normalized values for state variables:

$$x = \mathrm{RND}(0\sim1)$$

$$\frac{dx}{dt} = \mathrm{RND}(0\sim1)$$

$$\theta = \mathrm{RND}(0\sim1)$$

$$\frac{d\theta}{dt} = \mathrm{RND}(0\sim1)$$

Generate real values and limit them to between 20 % and 80% of the actual region

Using Fig. 5.2, get a control value (u) for these random values of the state variables

Fig. 7.2 Restart software in a simulated system model

Table 7.1 Typical state variable values for random state integers

Run row	State integer	Control value	Initial values of state variables			
	m	$U = \Phi(m)$	X_0	dx/dt_0	θ_0	$d\theta/dt_0$
1	113	−1	−0.153	0.016	0.017	0.054
2	6	1	−1.157	−0.002	−0.003	−0.408
3	2	1	−0.555	−0.186	0.129	−0.203
4	31	1	−1.087	−0.271	−0.133	−0.184
5	33	1	−0.065	−0.263	0.004	−0.391
6	1	1	−1.086	−0.136	0.012	−0.301
7	1	1	−1.070	−0.335	0.009	−0.188
8	94	−1	0.498	−0.185	0.004	−0.035
9	31	1	−1.066	−0.185	−0.115	−0.341
10	129	−1	0.604	−0.002	0.046	0.035

between ±0.17 rad, and $d\theta/dt$ (the angular velocity of the pole) between ±0.5 rad/s. It should be noted that while some state integers do repeat even in this small sample, the values of the state variables differ due to their random placement within each state. The values of ±1 correspond to LEFT and RIGHT.

Rows 6 and 7 of the table illustrate that repeated values of an initial state integer may be generated but the values of the state variables are quite different while still falling inside the same ranges. Figure 7.3 examines how the algorithm works for an example system integer of value of 113 after the algorithm unpacks its component state integers to $k_1 = 3, k_2 = 2, k_3 = 3$, and $k_4 = 2$ where k_1 is the integer region for $\boldsymbol{x}$ based on its boundary values and so on. Checking using Eq. 5.7 yields Eq. 7.1.

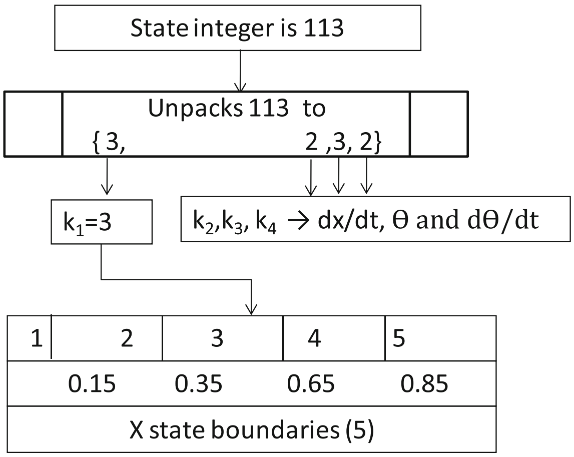


Fig. 7.3 Illustration of unpacking a state integer

$$\begin{aligned}
&Given \rightarrow k_1 = 3, k_2 = 2, k_3 = 3, k_4 = 2\\
&Given \rightarrow nb_1 = 5; nb_2 = 3; nb_3 = 5\\
&m = k_1 + nb_1 * (k_2 - 1) + nb_1 * nb_2 * (k_3 - 1) + nb_1 * nb_2 * nb_3 * (k_4 - 1)\\
&m = 3 + 5 * 1 + 15 * 2 + 75 * 1 = 113
\end{aligned} \tag{7.1}$$

Referencing Fig. 7.3, it is clear that the initial value of x must lie in region 3, dx/dt in region 2 etc. This implies that its normalized random initial value must lie be between 0.35 and 0.65 for the data shown. Equations 7.2–7.4 give the simple calculation required.

The key is to have the system generate a random offset within the real region of each state variable somewhere between its 20th and 80th percentile. If a zero random number is generated in Eq. 7.2, the offset *deltaX* is 0.2 which would be the lowest value. If the random number that was generated was 1, the offset would be 0.8. It was considered advantageous to keep the initial random values away from the state integer boundaries to ensure that the initial balancing task is always somewhat possible. Equations 7.2–7.4 illustrate the method.

$$deltaX = 0.2 + rnd(0) * 0.6 \tag{7.2}$$

$$X_{\mathrm{norm}} = boundary(k_1 - 1) + deltaX * (boundary(k_1) - boundary(k_1 - 1)) \tag{7.3}$$

$$X_{\mathrm{real}} = X_{\min} + X_{\mathrm{norm}} * (X_{\max} - X_{\min}) \tag{7.4}$$

For the data shown in Table 7.1 for the first row, where the system random number generated was 0.297, Eqs. 7.5–7.7 are identical to Eqs. 7.2–7.4 but with data values inserted. The value agrees with that given in the table in the fourth column for x_0.

$$deltaX = 0.297 \tag{7.5}$$

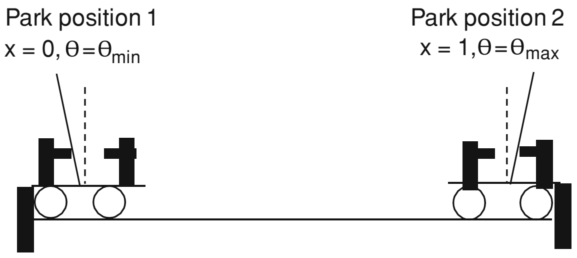


Fig. 7.4 Pole and cart park positions

$$X_{\mathrm{norm}} = 0.35 + 0.297 * (0.65 - 0.35) = 0.439 \tag{7.6}$$

$$X_{\mathrm{real}} = -1.26 + 0.438 * (1.26 - (-1.26)) = 0.153 \tag{7.7}$$

While this method can be used in simulation studies, it would be impossible practically speaking to actually impose these exact starting values in the physical system. It is also apparent that some of the control space especially near both boundaries of each region will never be accessed as a starting point for the controller. But it is still much more likely that a better coverage of the state space will occur using this method than simply releasing the trolley and pole from some fixed center position ($x = \mathrm{d}x/\mathrm{d}t = \theta = \mathrm{d}\theta/\mathrm{d}t = 0$) which coincidentally yields the state integer value of 113 every time. However at that moment, the motor will be running in one of two directions and the system will skid the trolley wildly in one of two directions before control can even begin.

7.3 Automated Catch up Restart Method

A second method is to use a catch up method of restarting the system. If the system is to use this automated restart process it must include a facility that parks the system at either end of the trolley's track, as shown in Fig. 7.4, before any simulated or actual run can begin. Identifying these two parking positions is essential if any form of automated restart is to be implemented.

To effect a system start up, the pole is moved under power away from its known current parking position by sending the appropriate motor control directive from the autostart subsystem which can be software or firmware. For example, if the system is at park position 1, the motor is given a RIGHT directive and vice versa. During this initial; motion the pole is simply *carried over the shoulder* of the cart. During this motion, the value of θ is either $\theta_{\max}$ or $\theta_{\min}$ depending upon from which rest position the system originates. $\mathrm{d}\theta/\mathrm{d}t$ and $d^2\theta/\mathrm{d}t^2$ are both zero as the pole is simply being carried by the trolley. Equation 7.8 which is derived from Eq. 5.2, shows the equation of motion during this initial phase of the restart.

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = \frac{(u.F)}{(M + m)} \tag{7.8}$$

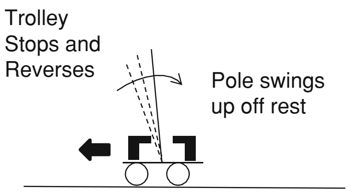


Fig. 7.5 Pole motion as trolley reverses

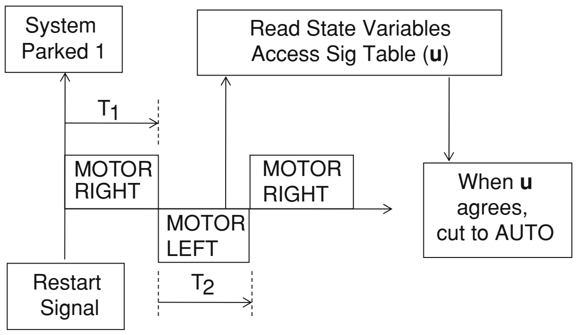

System Parked 1
Read State Variables Access Sig Table (**u**)
T1
MOTOR RIGHT
MOTOR LEFT
MOTOR RIGHT
When **u** agrees, cut to AUTO
Restart Signal
T2

Fig. 7.6 Timing diagram for catch up auto start process

After some short, but random time (T_1), the motor is abruptly stopped, and then reversed causing the pole to leave its rest position and start swinging up toward its central ($\theta = 0$) position. The motion now continues governed by Eqs. 5.1 and 5.2 using the initial values of the cart velocity as the initial velocity of the pole. Figure 7.5 shows this.

After another short but random time (T_2) or as soon as the pole swings past the origin ($\theta = 0$) the motor is stopped and reversed again, causing the pole to be jarred into free swinging motion.

Figure 7.6 shows a typical timing chart for the process in the case when the cart starts from park position 1. Immediately after the first pulse is applied to the motor, the BOXES software (or firmware) begins to collect data for the trolley and pole positions and velocities. During this process values of the system integer (m) are calculated and a corresponding value (u) from the signature table noted. As soon as the signature table value agrees with the autostart value, the system switches to AUTO mode and the control run begins. If for some reason this does not occur, after some preset time, the system reparks, based on the new orientation of the pole and the process repeated.

In this method, part of the BOXES algorithm previews the pole and cart as it is being started up but only takes control when the computed state integer reads a control decision from the signature table that agrees with the current actual motor state. Figure 7.7 shows a plot of the trolley position during a simulated autostart that uses two randomly generated timing values ($T_1 = 0.71$ and $T_2 = 1.24$). During the motor reversal, the system forces the trolley to stop ($\mathrm{d}x/\mathrm{d}t = 0$) before proceeding in the new value of direction. This method works in both simulations and in the actual physical rig described previously in detail in Chap. 6

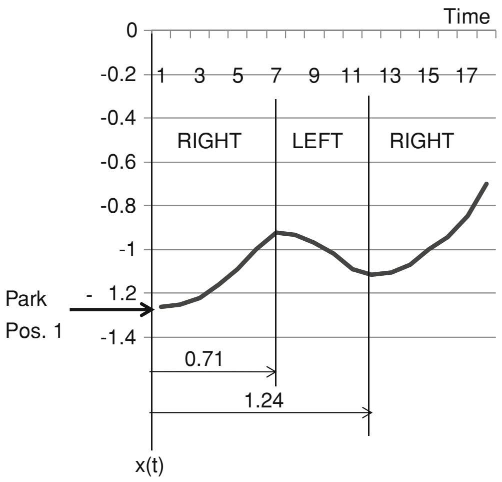


Fig. 7.7 Graph of trolley position during and after an autostart

7.3.1 Catch up Restart in Simulated Systems

In a simulated system, the software follows the same logic of Fig. 7.4 and randomly selects a parking position from which to begin its operations. The advantage of this method is that the simulation software is already running the model of the trolley and pole as it is forced to perform its random initial maneuvers.

Figure 7.8 describes the logic of this process which is akin to unhooking a glider from its towing aircraft after it has becomes airborne and is at a predetermined altitude.

Inside the algorithm, the value of motor direction is asserted until the state variables produce a state integer that indexes to the same value of motor direction as exists currently. When this occurs, the auto start is immediately disconnected and the system is allowed to bumplessly transfer into the control algorithm. At this point, runtime data starts to be collected and the system proceeds.

7.3.2 Catch up Restart in a Real Trolley and Pole Rig

In a physical system, the automated restart subsystem uses firmware that, at the start of a campaign or at end of every run, moves the pole to one of the park positions based on the orientation of the pole after its last move. If the pole is leaning towards the right, the system moves it to position 2 and vice versa.

To enable the physical system, as described in Chap. 6, custom electronics were designed and implemented to connect the trolley and pole rig to a mini-computer that performed all of the BOXES algorithm steps. One of these was an “autostart” facility that also enabled a manual controller to be added for testing and human

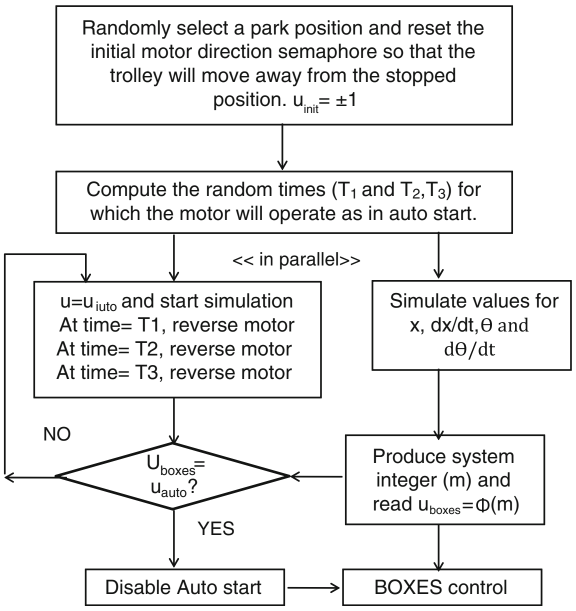

Randomly select a park position and reset the initial motor direction semaphore so that the trolley will move away from the stopped position. $u_{init} = \pm 1$
Compute the random times (T_1 and T_2,T_3) for which the motor will operate as in auto start.
<< in parallel>>
$u=u_{iuto}$ and start simulation
At time= T1, reverse motor
At time= T2, reverse motor
At time= T3, reverse motor
Simulate values for x, dx/dt, θ and dθ/dt
NO
$U_{boxes} = u_{auto}$?
Produce system integer (m) and read $u_{boxes} = \Phi(m)$
YES
Disable Auto start
BOXES control

Fig. 7.8 Algorithm for random catch up initialization

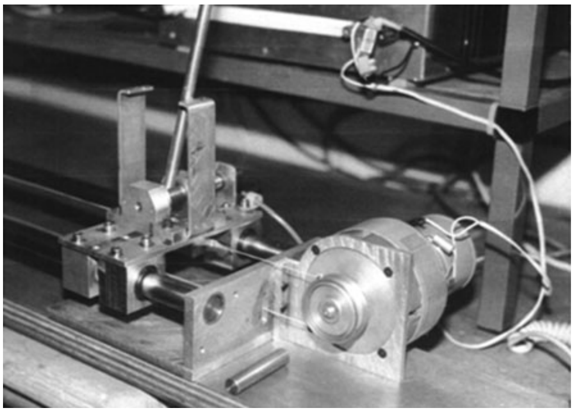


Fig. 7.9 Liverpool trolley with pole in start position #2

performance trials. Figure 7.9 is a photograph of the Liverpool trolley and pole in park position 2.

An outline of the logic of an autostart feature is shown as Fig. 7.10 and is intended only as a guideline for potential builders of a similar physical trolley and pole rig. As can be seen this is not a trivial task. In this design, Boyes [1] introduced the notion of a *sentry* firmware system that is used to detect any kind of out-of-range behavior of the system. If such an event occurs, it causes an immediate shut down of all the power to the trolley motor so avoiding damaging crashes.

Under such situations, the sentry compares normal signals to what is actually occurring and illumines the appropriate panel light(s) on the console to facilitate

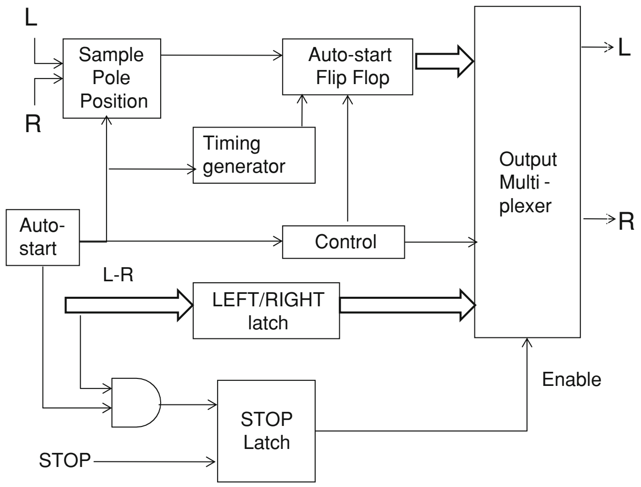


Fig. 7.10 Outline of autostart firmware logic

the tracing of the fault. In the event of a system failure, the sentry notifies the BOXES algorithm to cancel any further statistical database updates.

This essentially introduces a third state into the BOXES design. The three states in the real world system design are RUN, FAIL, and STOP.

7.3.3 Catch up Method Conclusion

Either auto restart procedures are invoked immediately prior to the start of any control run so that when the controller is set in motion it exists in a random state that is reasonable. This ensures that the controller can catch up with and seamlessly transfer operation into the auto mode. Should an autostart fail, the process is repeated until a good restart occurs with no accrual of statistics made during the start up process. This interface between the two phases of operation is crucial and was successfully implemented in the Liverpool experiments.

7.4 Systems Engineering

In simulation mode, the system can be programed always to start up in a controllable position within the limits of the state variables. To implement an autostart facility for a real system that utilizes a BOXES-type controller certain systems engineering rules are necessary and include:

- In the autostart mode, while the BOXES system is collecting data, processing it, and reading a control value from the signature table, but the motor on the trolley is digitally switched between LEFT and RIGHT by the startup firmware.

- A manual operator can override the system and perform control experiments that were useful in the initial training of the BOXES signature table.
- During the autostart sequence the BOXES processor might flag an out of range (FAIL) system integer ($m = 0$) which is caused by the rapid motor reversal process necessary to jar the pole from a rest position into motion. This FAIL mode must be ignored by the BOXES system.
- The BOXES generated value of motor direction may or may not agree with the current autostart value. If the two do agree, a decision to switch the system from AUTOSTART to the AUTO run mode can be enforced.
- Anytime, a third mode (STOP) must be enabled so that rapid removal of power to the rig is possible if any dangerous or damaging state or fault is detected.
- Learning is to be implemented only from after when the system is switched into AUTO mode. Any autostart data must be discarded from any statistical update procedures.
- The system is forced to move in two dimensions only using guide wires to prevent the pole from falling sideways when in motion. The trolley motion is bounded by two hard stops.
- Reasonable initial conditions must be generated so that the algorithm has opportunity to adapt and learn from each run.

7.5 Manual Experiments

It proved most useful to run the physical rig using the human computer interface which directly controlled the motor direction. Using this facility, it was possible to run the trolley up and down the track and to practice auto startup maneuvers to test the system electronics and failure mechanisms. Grant [2] suggests that a human-automaton collaboration can accelerate how the automaton can learn a task under the tutelage of a human, and conversely how an automaton can train another human to do the same task.

In the Liverpool system, the research team spent many hours running the system under manual control and became quite expert. As part of these experiments, novitiate users were asked to control the system and all usually failed within one second. The causes cited included a lack of concentration and the onset of panic as the pole began to fall. The untrained user performed the balancing task almost as well as a random signal generator that could be patched into the motor controller. Figure 7.11 is a graph of real data from the Liverpool experiment showing a comparison of random, (expert) humans and the BOXES method. The notion of merit was simplified for the human control runs to be an average time before failure occurred.

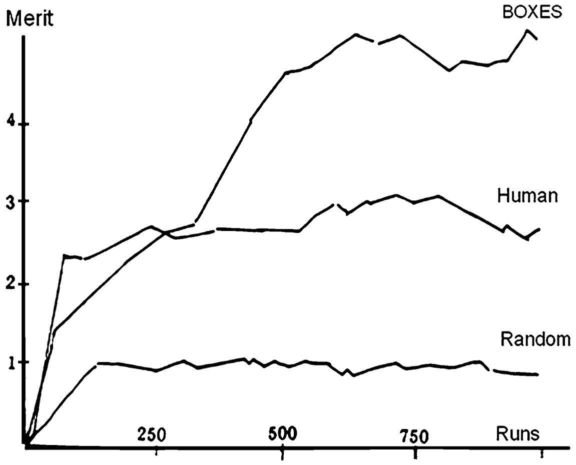


Fig. 7.11 Comparison of random, human and machine performance

7.6 Conclusion

The ability to auto start the system at the commencement of each run during a campaign of runs is essential. It is even more important that each run offers the BOXES system a different set of initial conditions so that the whole state space is exercised. Philosophically speaking, there are two possible learning strategies.

The first is to keep posing the same problem to the automaton until it has mastered that task. Once this has been accomplished, other tasks can then be set to extend the performance of the system. This is what happens if the trolley and pole system is started by letting go of a vertical pole near the center of the track. While useful, this is rather suspect as a learning path.

The second method that was adopted in the Liverpool work drove the trolley and swinging pole into some varying and random operating state and cut in the BOXES method only when a valid control state was recognized. This more refined method is slower and more complex to implement but disallows the creation of preferred states that would prevent the system from ever exploring the control space. It was important that the system operate in a fail-safe mode and that human operation be enabled to prove out the system electronics and firmware.

References

1. Boyes J. (1977) MPhil (CNAA) Thesis. Liverpool Polytechnic.
2. Grant, A.S. (1990) Modelling Cognitive Aspects of Complex Control Tasks. PhD Thesis Section 3.2 University of Strathclyde Department of Computer Science http://www.simongrant.org/pubs/thesis/3/2.html (Accessed April 1, 2011)

Chapter 8
Continuous System Control

8.1 Continuous Control

The application of the BOXES method so far has focused on a mechanically unstable system, namely the trolley and pole. Studies have been performed on several other systems such as multiple hinged poles driven by one trolley [1], backing up a tractor trailer [2, 3] and the like. These systems are in themselves quite challenging and the BOXES algorithm performs quite well in their control domain. What they have in common, that parallels board games, is that they are of relatively short duration and that they customarily fail repeatedly. This repetition, which is similar to playing a board game, allows the system opportunities to assess previous games and build upon a strategy for improvement.

The operating environment, particularly in simulations, is conducive to learning but in the real-world caution needs to be exercised in taking advice from an automaton. As the Liverpool experiments show, the system must be very carefully monitored so that disasters and human safety situations can be avoided. While the automaton may return control values based on mathematical correctness those values may be unreasonable, impossible, or even dangerous in the real world. However, having stated that, the automaton can perform many given tasks with undivided attention, repeatability, and great speed. Computer-based systems rapidly outperform most humans in almost any arena where learning occurs over many repeated excursions of the game.

Most engineering systems especially in the industrial world are continuous in nature. Examples of this are found in chemical process plants, biochemical reactors, and many more. The traditional approach to the control of systems such as this is to create a model based on physical laws and any known dynamic behaviors of the system. Once such a model is devised, a control law can be synthesized using a variety of methods such as PID [4] if the system needs to be tuned, state space [5] if linearization is possible, or Rosenbrock's method [6] if the system is nonlinear. Control is accomplished based upon computed values of actuator 'settings being passed back to the process using digital to analog conversion.

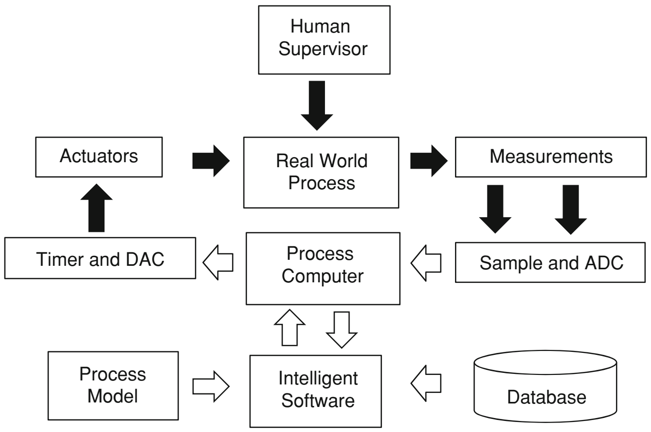


Fig. 8.1 Intelligent process control

In Fig. 8.1 the black shaded arrows depict the flow of real-world analog data to and from the plant and the unshaded arrows show digital data.

In any DDC (direct digital control) plant, with or without the intelligent components,it is essential that real data from the plant instrumentation must be made available to the controller in a timely manner. In return, the process computer must supply a valid control signal to the plant in accord with the plant's time constant. If the intelligent component of the control software involves multiple iterations or other time consuming algorithms, a mechanism for asserting a "best value so far" must be included in the real-time section of the program. This implies that a valid, even if it is only a default, value of control signal must always be supplied to the DAC (digital to analog convertor) at the correct moment. With the increased processing speed of modern processors this is less of a concern than in the past.

Also, the size of the actuation signals may be amplified or attenuated inside their control mechanisms for process or safety reasons. For example, in a bioreactor, it may be imperative that the feed rate be restricted to some maximum value during the preliminary germination of the enzymes. Once growth is established, the feed rate may be increased to some other known limit. If the process control system fails to comply with these fermentation regulations, a human operator may need to intervene and correct the control valve setting. When this happens, the process computer will not necessarily know that its last calculated value was overridden and essentially ignored. Such discontinuities cause real problems in DDC systems.

Another obvious concern with such classical methodologies is that they are based on models of the real processes and as such must reconcile actual and idealized data and make control determinations based on the result. With the rather checkered history of software failures, there is much debate over whether primary control responsibility can ever be assigned to an intelligent system, and the architecture of Fig. 8.1 implies that the intelligent component really only refines or optimizes what may be a simple primary control scheme. Software bugs in real-time systems can be very subtle but have deadly results. It was reported [7] that three people were killed and three others critically injured when a radiation therapy machine malfunctioned and delivered lethal radiation doses to patients. The cause was a subtle bug that created a race condition in the system's control logic that allowed a technician to inadvertently configure the equipment in such a manner that the electron beam fired in high-power mode without the proper patient shielding. Industry is very wary of computer controlled systems, and with some justification.

Process systems must run flawlessly for long periods of time and failure is not an option. With this in mind, the remainder of this chapter is devoted to a discussion of how continuous systems may be designed without any prior knowledge of the system's dynamics using what is called black box control [8] such as that of the BOXES method. In order for this to be a possibility, it will require redefinition of the term failure.

8.2 A Different Perspective on Failure

This section describes two aspects of failure which support the notion of failure assisted methodology with the caveat that the resultant A.I. contribution may augment a simple control schema rather than be expected to assert primary intelligent control. Rather than having to design control laws based on the internal dynamics of a system, the BOXES approach [9] will provide a method in which observable outcome-based control is achieved.

8.2.1 Constructive Failure

This responds to the ideology that failure carries with it the unfortunate connotation of helpless defeat. Failure is an integral part of the learning process. Failure in this sense is to be considered in the more positive light of a mechanism to allow reflection on one learning situation so that choices be made that can better perform the task in hand. Systems that use statistics especially in which initial random decisions are refined by using data captured during (many) processing runs, must pause after some interval of time in order that those weighting factors and/or other statistical factors can be recomputed. These events must occur at some regular interval. In the trolley and pole system, these deliberations occur after the system failed, whereas in continuous systems such learning mechanisms can only take place on a controlled event basis. It may be that these events are self-induced, for example, using another type of timeout clock interval generated solely for the purpose of learning. This can take advantage of the disparity between real-time sample rates and other algorithmic throughputs. The method may also take advantage of process generated events, for example, if a variable strays outside of a control limit which may convey an advance warning of an impending instability. If the system is shut down temporarily, for example by an operator, the automaton can opportunistically update its learning data during that downtime. Corrective actions, induced by a failure such as this, are intensely valuable in the production of a balanced, systematic adaptive strategy that is designed to cover all eventualities.

8.2.2 Limited Failure

Failure is unacceptable in any context if the resultant effect would cause risk or harm, or if the system under control is driven into some impossible situation. The first condition is obvious; to destructively test a system while still learning is simply nonsensical. Yet, during control excursions, learning controllers that update statistics have little time for *reflection*, the emphasis being on rapid *reaction*. It is because of this if the system fails to detect an impending massive instability, that failure is truly catastrophic. However, Berztiss [10] comments on how a catastrophe may be avoided by allowing the system to operate for a while in an unsafe state! In this regard, failure exonerates itself.

The second situation may occur more regularly than is first thought. If the driving task presented to the system is not feasible within computational, or physical, limitations, then the learning algorithm should disqualify the results of that run from influencing the statistical accumulators—in order to avoid mathematical or control havoc. It is apparent that a learning system must include some upper-bounded quantification of the *difficulty* of the task in the statistical allocation of penalties and rewards. This transforms into the paradox of the need for the simultaneous existence of a *demanding taskmaster* and a *compassionate critic*. The instructional agent must rigorously test and evaluate the system and yet sympathetically take cognizance of difficulties in the variable problem domains. The degree of difficulty of the task can be gradually increased as the level of learning and goal attainment increases. This gradation of challenge assists the algorithm in finding a smooth learning curve.

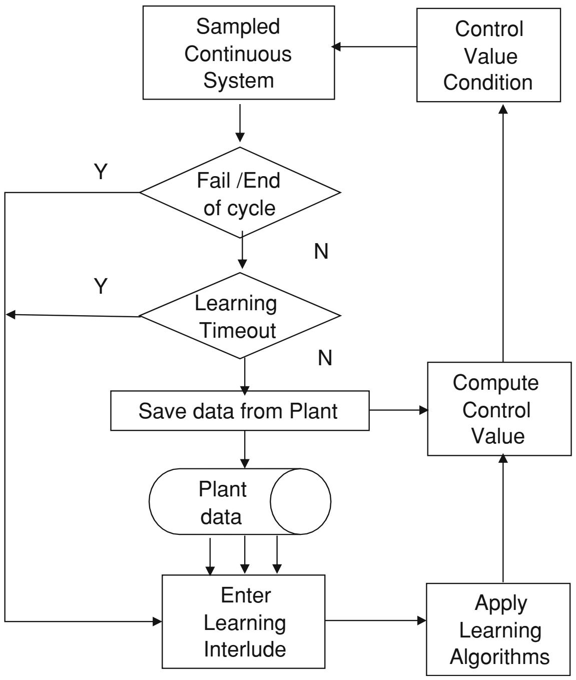
8.2.3 Creating a Learning Interlude

In the attachment of any learning actor to a continuous system the algorithm must be given sufficient time to process current data and formulate a strategy for improvement. This conceptually could occur during every sample within the DDC schema or after some preset number of samples. What is important in either case is that sufficient data is allowed to accrue and enough actuation time given for any applied changes to be effective. If a system fails or is operating in cyclic mode (shown as case 1 in Fig. 8.2) or fails (as shown as case 2 in Fig. 8.2) the learning

SAMPLE INTERVAL NUMBERS – CORRESPONDING TO REAL TIME														
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CASE 1: SYSTEM CYCLES EVERY 7 SAMPLES														
>	>	>	>	>	>	XX	>	>	>	>	>	>	XX	>
CASE 2: SYSTEM FAILED AT INTERVAL #10														
>	>	>	>	>	>	>	>	>	XX	??	??	??	>	>
CASE 3: CREATE LEARNING INTERLUDE EVERY 5 SAMPLES														
XX	>	>	>	>	XX	>	>	>	>	XX	>	>	>	>

Fig. 8.2 Learning interlude intervals

Fig. 8.3 Learning interlude options



interlude can be inserted before the next sequence begins. If the system is truly continuous, an artificial interlude must be created as shown in case 3 in Fig. 8.2 which is a comparison between the options indicating timing constraints.

Obviously if the sample interval is very short, the learning interludes which are marked in reverse white/black as “XX” in Fig. 8.2 would need to be short also and may prove untenable for uninterrupted DDC. If the system is fairly slow, the learning interlude may be comfortably completed within the DDC control sample interval. Control action intervals are marked “>” but if the system fails, the lag in restarting (shown “??”) is dependent on how any restart feature is enabled and may cause the

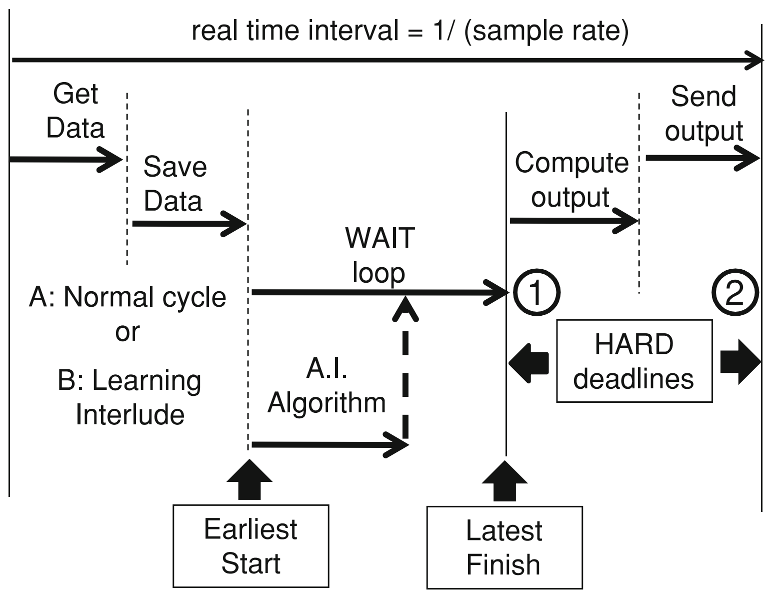


Fig. 8.4 Software components inside a sample

plant to shut down. Figure 8.3 illustrates a software structure showing the timing conditions under which safe operation and learning can occur in a collaborative environment.

At a more granular level, Fig. 8.4 illustrates the timing logic of the software components inside any sample interval for the AI attachment to be feasible. It is clear that the process's hardware and clock speeds largely determine the duration of the data acquisition and control output intervals, and an acceptable *WAIT* process that can vary in order to fulfill the requirements that the two hard deadlines impose.

In the event that a learning interlude becomes active for any of the reasons given above, there are two software issues to be considered. First the learning agent cannot start before the *EARLIEST START* signal occurs, and must complete before the *LATEST FINISH* semaphore is exercised. If the algorithm fails to complete within this time interval, all data changes are ignored and the system continues with existing control values. If the learning algorithms do complete in time and the control data updated, the system must re-enter the regular *WAIT* agent until the first hard deadline occurs.

8.3 Outcome-Based Performance Evaluation

To reduce processing time inside the AI learning module, the BOXES system needs to accomplish outcome-based learning which is less quantitative than traditional computationally intensive methods. It is suggested in this chapter that

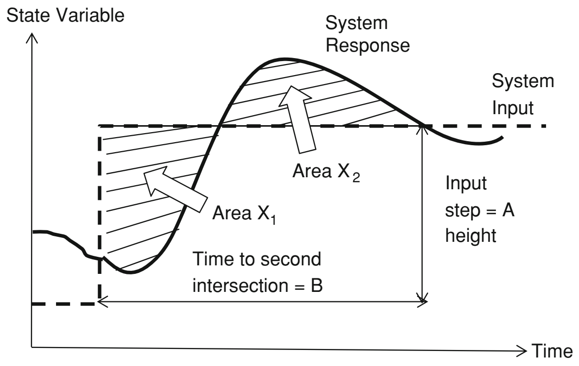


Fig. 8.5 System response curve

AI augmentation of existing controllers may be an area in which the BOXES algorithm might prove very useful. Rather than giving the learning system the task of minimizing a cost functional based on what is often a complicated state space method, it is proposed here that the dynamic performance of a process can be evaluated in a more qualitative manner, such as how well oscillations are being suppressed, or how far above the signal did the process overshoot.

Figure 8.5 shows a typical response of a system of some order to a step input showing a delay in reaching the demanded set point and the transient oscillations that occur. If the controller was to be assessed on its performance in a traditional setting it would require an analysis, such as a Laplace Transform of the system's differential equations and the offset step function input.

8.3.1 An Example Outcome-Based Approach

If the system was impossibly perfect, X_1 and X_2 in Fig. 8.5 would both be 0 and the output would follow the input, which in the BOXES notation would constitute a perfect game. If the system ignored the input, perhaps because of gross over damping, and just continued from its present value, provided some preset time consideration was set for **B**, the worst value of the area would be A * B.

This presents an opportunity for a BOXES algorithm to compute a value for response *merit*, which is familiar to the method. Equation 8.1 is an example of such a method.

$$\mathrm{Merit} = 100 \times \left\{ \frac{1 - (\mathrm{X1} + \mathrm{X2})}{\mathrm{A} \times \mathrm{B}} \right\} \tag{8.1}$$

The perfect game, in which $X1$ and $X2$ are both 0 would produce a merit of 100% and the worst case, when $X1 + X2$ is equal to $\mathrm{A} \times B$ would yield a 0 merit value. Using an approximate curvilinear squares method, the response shown in

Fig. 8.5 has a merit of around 62%. It would be the task of the learning algorithm to increase the overall merit value of the system over time and to punish and reward control values within the application space. The state space would be divided into zones for each state variable, and state integers assigned. The state integers would be combined into a system integer and a control feature asserted based on the signature table value corresponding to the system integer in an identical manner to that used in the trolley and pole.

8.3.2 Outcome-Based Assessment

It is apparent that there might be some questions as to the prudence of using how a system reacts to certain input stimuli as a means to effecting control laws. There is a lingering uncertainty that there is no guarantee that there might not be an inherent instability within the dynamics of the system. However, in the case of systems that are ill defined mathematically, or so nonlinear that computational methods are too time consuming in the real world, the BOXES method may be the only viable control option.

8.4 Training Continuous Automata

In almost all learning systems for use in real-world applications, the designer is always faced with the elusive problem of knowing when training is complete and actual mechatronic implementation should begin. This is particularly problematic in continuous systems. If the learning algorithm includes any notion of long-term statistical inference then the significance of an early occasional poor or extraordinarily successful run may be grossly exaggerated. Kubat and Zrizakova [11] postulated wisely that *without forgetting, there is no learning.* One technique that BOXES incorporates is to introduce a *forgetfulness* factor that, by repeated application after each run, successively down-rates the influence of past data. This ensures that the learning process remains bounded yet online and live, and that control actions are never stranded in some stagnant solution space.

In systems of the type described in this chapter long-term swings in process behavior are accommodated by a natural migration of the solution vectors in concert with emerging trends. It is because of this that the system continues to learn throughout the lifetime of the application and adapts to physical component wear, accidental conditions, and other ill-defined changes in the process. Perhaps the feature that is most overlooked is the vast number of *fail and learn* processes that the BOXES algorithm can complete inside the sample interval of the system. In a typical industrial process the time constant is measurable and can operate successfully with sample intervals that are very long, maybe of the order of minutes, compared to execution times in a modern processor. There is usually time to fail, re-group, and try again.

8.5 BOXES Control of a Continuous System

There are many ways of controlling systems provided that they supply timely and measureable outputs in response to control actions. As mentioned above, PID and State Variable are two such popular methods. Both methods require that signals be sampled at an appropriate frequency, and that control decisions be made within a finite time window. It is the former method that will be shown to benefit from BOXES augmentation.

8.5.1 PID Control

A system can be controlled by using the error signals constructed from the difference between the desired and actual levels of the variables. The necessary control signals (c_i) can be generated from calculus-based operations on the error signal using Eqs. 8.2 and 8.3

$$e_i = \mathrm{Yref}_i - Y_i \tag{8.2}$$

$$c_i = KP \cdot e_i + KI \cdot \int e_i \cdot dt + KD \frac{de_i}{dt} \tag{8.3}$$

where, KP, KI, and KD are tunable parameters that determine the span of control effectiveness of the PID arrangement. This method does not require knowledge of the system model, and is only optimal in the sense of its control over the shape of the output waveforms.

8.5.2 State Variable Control

Modern, optimal adaptive controllers utilize the state variable approach to manipulate matrices that mathematically identify a model of the plant into cost functional terms. These expressions are then minimized or maximized to produce an optimal control strategy matrix. The model may contain matrix elements that are nonlinear and/or time-dependent that serve to further complicate the algorithmic procedure. Detailed knowledge of the model of the process or plant is essential for a successful calculation of the control matrix which would seem to eliminate it from being a candidate for BOXES control.

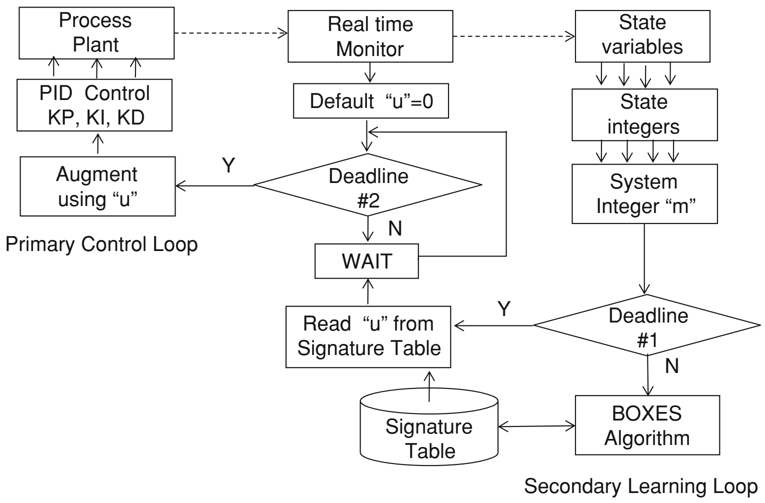


Fig. 8.6 A hybrid adaptive controller

8.5.3 BOXES for Continuous Systems

The short duration of the balancing runs of the trolley and pole are easy to relate to Michie's idea of a control game. In order for the BOXES method to be applicable to continuous systems, it is apparent that uncertainty of the signature table values could not be tolerated in primary control loops, but that it might be possible to attach control values from the BOXES method to a stable controller. Russell [12] reported that the augmentation of a conventional control schema, such as Proportional, Integral, and Derivative (PID), by BOXES-generated contributions, read from a signature table, did show evidence of dynamic tuning of the system to be in effect. Figure 8.6 illustrates such an arrangement which is designated a hybrid adaptive controller.

The BOXES augmentation is attached to the controller by modifying the KD and KI terms in Eq. 8.3 using the values of u stored in the signature table matrix. The method will seek to create and maintain control *intelligence* in the control matrix (ST) that contains the set of u values being trained to effect control.
If,

ST	Current BOXES signature table control matrix
m	Current system integer value
u	Current decision value for state $u = \mathrm{ST}\ (m)$
KP	PID gain parameter—left unaltered
KI	PID integral term tunable parameter
KD	PID derivative tunable parameter
a,b,c,d	Augmentation share factors $0 < value < 1$.

Table 8.1 Augmentation span of KI and KD

	Proportional factor, KP	Integral factor, KI	Derivative factor, KD
No augmentation $u = 0$	KP constant	KI $= a + b/2$ constant	KD $= c + d/2$ constant
Positive augmentation $u = +1$	KP constant	KI $= a + b$ increased	KD $= c$ reduced
Negative augmentation $u = -1$	KP constant	KI $= a$ reduced	KD $= c + d$ increased

The goal is to seek values of u that can modify the PID parameters to encourage cooperative learning in the Integral and Derivative terms. The KI and KD terms have very specific control significance related to response rate, steady state error, etc., and as such introduce two stratagems into the game. One typical attachment is contained in Eqs. 8.4 and 8.5:

$$\mathrm{KI_{aug}} = \mathrm{KI} \times \{a + \frac{b}{2}(u + 1)\} \tag{8.4}$$

$$\mathrm{KD_{aug}} = \mathrm{KD} \times \{c - \frac{d}{2}(u - 1)\} \tag{8.5}$$

Table 8.1 shows how the KI and KD terms are modified when a non-zero value of u is applied to each equation. When no augmentation is applied, the values of KI and KD are fixed based on the values of $a,b,c,$ and d. If the BOXES attachment value for u is +1, the integral coefficient is increased while the derivative term is reduced. The opposite happens if the value of u is -1. In this example, the gain parameter (KP) is unaltered.

At the end of each controlled-learning time-window, a post-processing algorithm evaluates the previous run in terms of success in achieving the overall goal of the system. Each cell in the control matrix is evaluated based on its contribution to the prior run and its control value updated where necessary. Section 8.6 is an example illustrating how the system works for a simulated second-order plant. The plant model was incidental and used solely to create data values to populate state variables.

8.6 A BOXES Augmented Controller Results

The response of a continuous second-order system to unit step impulses was selected to illustrate the viability of the method and to clarify the type of optimization that can be expected. The PID augmentation described above was used and the process was simulated by the Laplacian given in Eq. 8.6 with k as some constant.

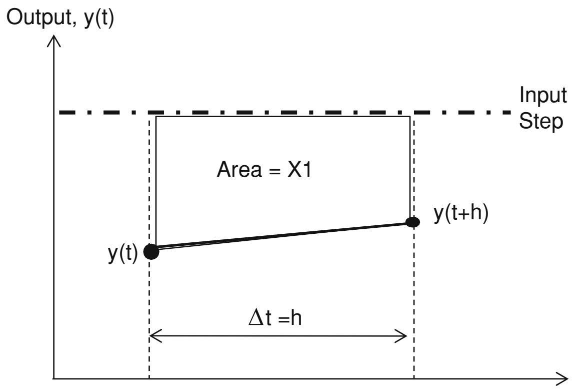


Fig. 8.7 A slow response transient

$$\text{Process Plant} = P(s) = \text{K}/s(s+k) \tag{8.6}$$

The BOXES algorithm was presented with sampled values of the output variable and its velocity. The performance of the system was adjudicated by the divergence of the output waveform from the input step value.

8.6.1 Outcome-Based Reward and Penalty

In this example, the divergence from the ideal was measured as the area enclosed by the response and the step input. The difficulty of the task was defined as the effective step size—which is the difference between the step height and the initial condition of the system. The failure window was preset as a fixed time step (h). The computation of appropriate rewards and penalties in this instance are a matter of simple trigonometry. Figure 8.7 shows the cases when the system response (y) has yet to cross the desired unit step level in a given time interval (h). The state variable(s) define the system integer as described throughout the text, and the BOXES automaton is assumed to have applied its PID augmentation value (u).

The area between the input step and the transient which is assumed linear between $y(t)$ and $y(t+h)$ reflects the failure level of the system in playing the perfect game. Using Eq. 8.1, with $X1$ being the area of the shaded section of the graph and $X2$ zero, the value of divergence that can be assigned to the cell can be shown easily to be given by Eq. 8.7.

$$\text{Divergence}(m) = 100 \times \left\{ \frac{1 - 0.5 \times (y(t) + y(t+h))}{h} \right\} \tag{8.7}$$

Figure 8.8 illustrates the situation when the system is faster and the output crosses the desired unit step input during the small time interval (h).

The divergence for the cell is calculated by Eqs. 8.8 and 8.9 as follows:

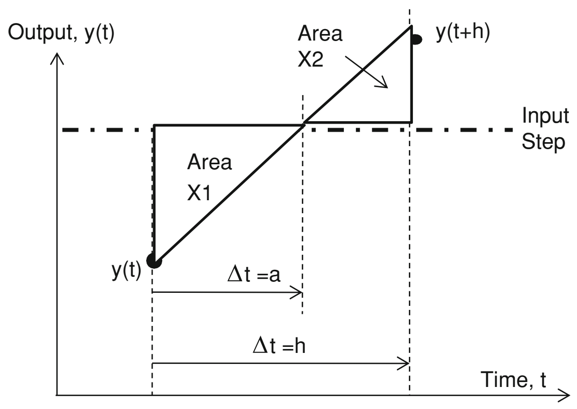


Fig. 8.8 A fast response transient

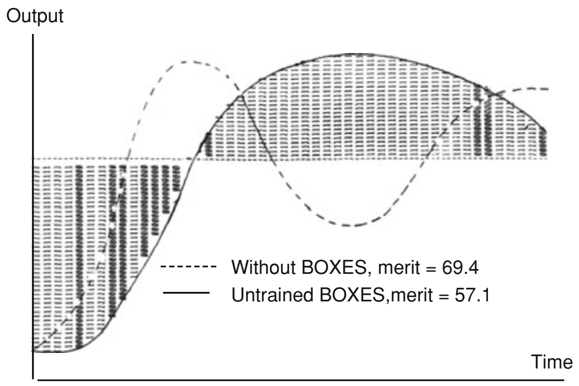


Fig. 8.9 Response with untrained BOXES augmentation

$$\text{Area of triangle, } X1 = 0.5 \times \{1 - y(t)\} \times a \tag{8.8}$$

$$\text{Area of triangle, } X2 = 0.5 \times \{y(t+h) - 1\} \times (h - a) \tag{8.9}$$

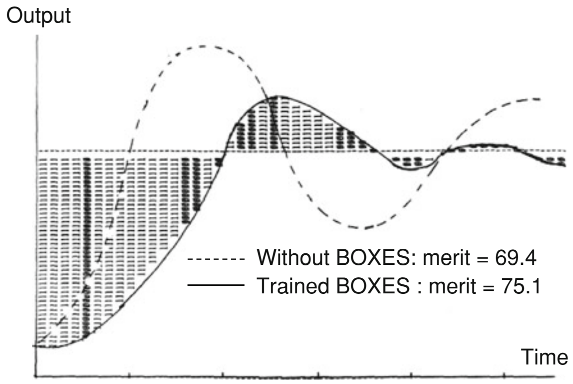
where, a is the time inside the step where the system output crosses the unit step input, and using similar triangles is given by Eq. 8.10.

$$a = h \times (1 - y(t)/(y(t+h) - y(t)) \tag{8.10}$$

Combining Eqs. 8.8, 8.9 and 8.10 gives the cell divergence given by Eq. 8.11.

$$\text{Divergence}(m) = 100 \times \left\{\frac{1 - 0.5 \times (X1 + X2)}{h}\right\} \tag{8.11}$$

Fig. 8.10 Response after trained BOXES augmentation



8.6.2 Results Obtained for the Example

Figure 8.9 shows the effect that an untrained BOXES augmentation has on the simple second-order system response to a step input.

Figure 8.10 shows the response of the same system after a period of BOXES training. This indicates how the BOXES algorithm aggressively learns to reduce transients by augmenting the PID parameters. Initial observations of the response shapes showed that the algorithm adjusts both the amplitude and frequency of the oscillations in its search for a perfect solution which would be indicated by merit values approaching 100.

8.7 Conclusions

This chapter has illustrated how the BOXES method can be applied to real-time, real-world control systems. It showed how apparent failure can be turned to positive advantage. The avoidance of disastrous decisions is buffered by the concept of augmentation and never-ending adaptive learning is allowed through timeout windows. Most industrial systems fall into the self-organizing category [13] but it is only by careful modeling and linearization that state variable adaptive controllers can be considered viable. It seems fallacious to then seek optimality using a non-exact and simplistic model.

Hybrid adaptive controllers, in which simple, but adequate, control schema are augmented by black box, heuristic paradigms, such as the BOXES method, effect real-world control without any a-priori knowledge of the dynamics of the system being controlled. The BOXES method takes an initially random control matrix that is modified by repeated statistical inference until a *box* gathers enough experience to become assured of its (binary) value. The initial matrix does influence the training time, but is preferable by far to any form of rule-based initialization.

References

1. Wieland A (1991) Evolving neural network controllers for unstable systems, IJCNN-91-IEEE Seattle International Joint Conference on Neural Networks II: 667- 673.
2. Woodcock N. et al. (1991) Fuzzy BOXES as an Alternative to Neural Networks for Difficult Control Problems. *Applications of Artificial Intelligence in Engineering VI*. CMP Elsevier Press Essex UK 903-919.
3. Nguyen D H and Widrow B, 1990 Neural Networks for Self-learning Control Systems. *IEEE Control Systems Magazine*, April 1990: 19-23.
4. http://www.expertune.com/tutor.html
5. http://www.scholarpedia.org/article/State_space
6. http://www.applied-mathematics.net/optimization/rosenbrock.html
7. http://www.devtopics.com/20-famous-software-disasters
8. Canudas de Wit C A. (1988) *Adaptive Control for Partially Known Systems*, Elsevier Press, Amsterdam: pp. 99.
9. Russell, DW (1994) Failure Driven Learning in the Control of Ill–defined Continuous Systems *Int. Journal of Cybernetics & Systems.* Taylor & Francis. Washington DC: Vol 25: 555-566.
10. Berztiss, A T. (1993) Catastrophe Prevention in Safety-Critical Systems*Proc. Fifth International Conference on Software Engineering and Knowledge Engineering-SEKE'93*, San Francisco, CA, Knowledge Systems Institute, pp. 102-108.
11. Kubat, M & Zrizakova, I. (1992) Forgetting and Aging of Knowledge in Concept Formation. *Applied Artificial Intelligence*. Hemisphere Publishing Company New York Vol. 6:2 pp. 195.
12. Russell, DW (1992) AI Augmented Process Control *Artificial Intelligence in Real-Time Control 1991 (ed. Rodd., M.G and Suski, G.J.) Proc. 3rd. IFAC Workshop on AI in Real–Time Control, AIRTC-91,* Sonoma, CA, IFAC Workshop Series No.5. New York, Pergamon Press, pp.75-79.
13. Ashby, W.R. (1956) *Introduction to Cybernetics*, Chapman and Hall London.

Chapter 9
Other On/Off Control Case Studies

9.1 On/Off Control

On/off switches are the simplest control mechanisms. They can redirect components on a conveyor belt, turn a valve to divert fluids between two pipes or channels, or even control the supply of primary power. In real systems, this is not quite as easy as it sounds because the dynamic effect of turning almost anything on or off effectively injects a step input into the system which may create a whole other set of dynamic behaviors which can induce instability or even chaotic transients. An elementary explanation for this is seen by considering the Laplacian for a step function which is 1/s. When such an input is attached to a dynamic system, the order of the system is increased which in turn may cause oscillations and other deleterious effects.

In this chapter, three systems are described in some detail to explain how the BOXES methodology may be an effective alternative to the more traditional control methods. Figure 9.1 illustrates the commonalities between all three systems to be considered in regard to how the BOXES method can be applied.

In Fig. 9.1 the system integer is shown as *m* and the corresponding signature table value as *u* or $\Phi(m)$. Data that will support the learning and update processes is collected every time a new cell is entered and exited. Whenever the current run ends, either because of some time event or after an out of range condition, the system integer is set to zero forcing the update process to be invoked.

9.1.1 Learning Algorithms

In any BOXES implementation it is necessary to construct some measure of system outcome or performance metric which has been labeled *merit* throughout the book. In prior examples the trolley and pole system strove to lengthen the overall run time. In the continuous second order system, perhaps the area under a curve could be minimized. Each of the three examples will require some similar feature.

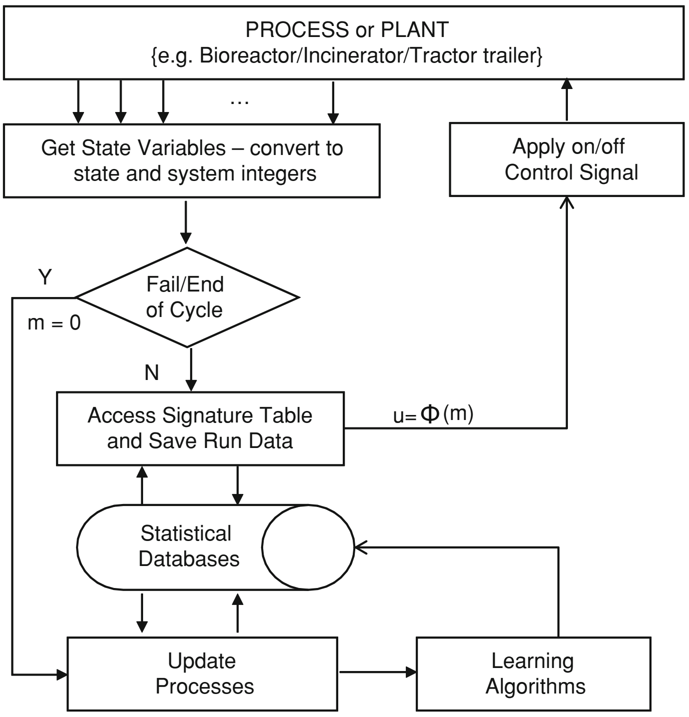

PROCESS or PLANT
{e.g. Bioreactor/Incinerator/Tractor trailer}
…
Get State Variables – convert to state and system integers
Apply on/off Control Signal
Fail/End of Cycle
Y
m = 0
N
Access Signature Table and Save Run Data
u=Φ(m)
Statistical Databases
Update Processes
Learning Algorithms

Fig. 9.1 Commonalities in BOXES applications

Once this has been established it is possible to create a desired level of effort and rate of learning that will be needed in the update phase. Each application must have an overarching game plan in which changes in merit are reflected in the strength of each decision in the signature table.

9.1.2 Run Time Data

In any BOXES system, it is necessary to save at least one data item for each cell. It is customary to always use the number of times any particular cell is entered because it reflects the level of participation and responsibility for the current run. Other data might be a shortness of path, summation of time of entry, efficiency, cost etc.

9.1.3 Signature Table Update

In order for the signature table to reflect the individual contributions to the overall game plan, it is necessary to compute pairs of decision strength values corresponding to each decision option. Conceivably, the signature table can contain any number of values and would then have to have that many strength values, but it would then not be a bang-bang system.

During the update phase, all data is aged using a forgetfulness factor and a new value of global merit calculated. Data from the last run is merged on a cell by cell basis with previous data that is tagged with the current decision value. The individual strengths of each decision for each cell are then calculated and the signature table value altered based on the greater strength value. Section 5.5 described this process in detail for the inverted pendulum system.

9.1.4 Application Summary

Models are supplied for each of three example systems and the reader is encouraged to expand on these or use one of the many alternate models that are available in the literature. The bioreactor model is covered in greater detail than the others as a benchmark of how the other systems can be studied. In reality, the BOXES controller would be trained by multiple excursions of the models before being considered suitable for application in the real world.

9.2 Fedbatch Fermentation

The fermentation process has been the topic of much study [1–3]. Each of these and many other models are usually expressed by a set of well known nonlinear differential equations which forms the mathematical basis for the design of control algorithms. However, experimentally, the models of the process are very difficult to validate. In large industrial plants, biomass growth is very sensitive to even the slightest variance in its environment and subject to rapid failure. Another reason that fermentation is difficult to model lies in the non-Newtonian nature of the fluid–solid mix that forms in the reactor. If the state variable method is selected as the control paradigm, the approximated state equations must be linearised in order to form control laws. It is not surprising that only limited success is to be expected [4]. To avoid the linearization process, Queinnec [1] suggested that Rosenbrock's nonlinear programing method [5] could handle the formulation of an optimal feed pump schedule which is the normal control mechanism. This method attempted to adjust the nutrient flow in such a manner as to sustain the maximum rate of substrate consumption at all time. These methods all, and quite rightly so, use the mathematical model in the production of a control signal. While analytically supportable, the feed pump schedule (see Fig. 9.6 later) seemed impractical to follow.

A further complication is that in an online environment all system variables are first sampled and then preprocessed into state variables before being used in the control algorithm. Sampling theory indicates that this always adds another level of

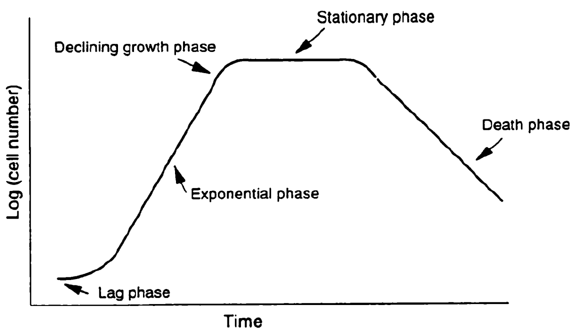


Fig. 9.2 Fermentation growth phases—used with permission

complexity and possible instability into what may already be a poorly defined system.

9.2.1 The Fedbatch Fermentation Process

The fermentation process occurs in phases and the success of a batch which is measured by the quantity of product is strongly related to how the process is managed in terms of initial seeding and feeding strategy. Figure 9.2 shows the five phases of product growth. The control game is to defer the onset of the "death" phase by carefully promoting the production phases.

A typical industrial fedbatch fermentor [6] system is shown in Fig. 9.3.

It is customary to use the nutrient feed pump (shown as Q in Fig. 9.3) as the primary control agent as this has the most effect on the fermentation process. The scheduling of the pump sequences is not without constraints.

9.2.1.1 Volumetric Limitation on Feed Rate

As the nutrient required to support the fermentation is fed into the reactor, so the active batch volume obviously increases. For any finite size reactor, this volumetric constraint forces control to be subject to an *over-ride* condition. Whenever the maximum volume is reached, the feed pump must be switched off regardless of any control requirements. This introduces a bounded temporal region of control that is shorter than the batch fermentation time. Once the reactor is full to volumetric capacity the fermentation process continues without further control.

9.2.1.2 Computational Limitations on Feed Rate

What is less obvious is that a control algorithm may erroneously compute an impossibly large or negative value for the nutrient flow rate. In the latter case, a zero value lower-limit should clearly be imposed and at the other extreme the

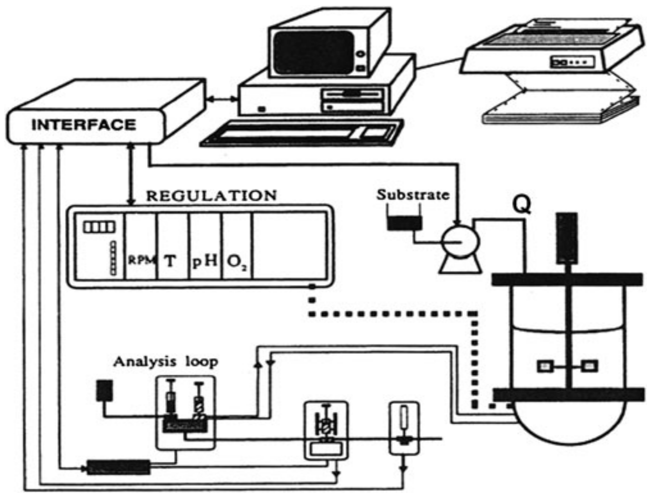


Fig. 9.3 A typical [1] fedbatch fermentor—used with permission

maximum allowable feed rate which is largely determined by the rating of the feed pump.

Again, this causes the actual calculated control value to be subject to override and introduces regions of time in which the controller is to be rendered impotent as illustrated in Fig. 9.4.

9.2.1.3 Nutrient Feed Pump Schedule

The nutrient can be supplied in a variety of modes: continuously at some computed rate; with an ON/OFF schedule; or in a combination of both. The decision as to when and how to inject nutrient can be time or state driven as will be seen later. The requirement is to enact a feed pump schedule that will cause the maximum quantity of product to be formed in the minimum time, and with minimum cost.

9.2.2 A Fedbatch Fermentation Model

The following constitutes a mathematical model of the fedbatch fermentor [1] and is based on the principles of dynamic material balance.

9.2.2.1 A Fermentor Model

Equations 9.1–9.7 show the relationship between the fermentor variables and describe the growth rate of the biomass (X) and the formation of product (P) in response to feeding nutrient at the rate of Q liters per minute.

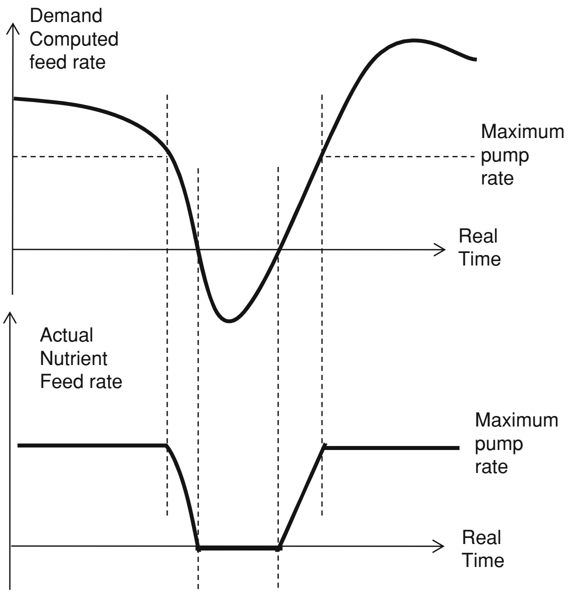


Fig. 9.4 Nonlinearity in the actual feed pump schedule

$$dX/dt = \mu * X - X * Q/V \tag{9.1}$$

$$dS/dt = -v_s * X - (S - S_{in}) * Q/V \tag{9.2}$$

$$dP/dt = vp * X - P * Q/V \tag{9.3}$$

$$dV/dt = Q \tag{9.4}$$

Ghose and Tyagi's [6] model is used to describe the specific rates:

$$\mu = \mu_{max}(1 - P/P_m) * S/\left(K_s + S + S^2/K_i\right) \tag{9.5}$$

$$v_p = v_{max} * S/\left(K_{1s} + S + S^2/K_{1i}\right) \tag{9.6}$$

$$v_s = v_p/Y_p \tag{9.7}$$

Where

X	Biomass concentration (g/l)
S	Substrate concentration (g/l)
P	Product concentration (g/l)
V	Working volume of the fermentor (l)
Q	Rate at which nutrient is fed to the fermentor (l/min)
μ	Specific growth rate
v_s	Specific rate of substrate consumption
v_p	Specific production rate

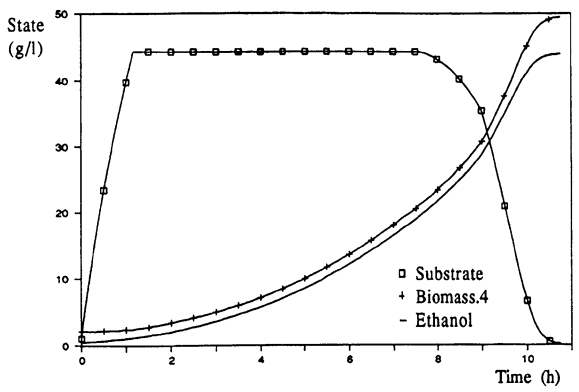


Fig. 9.5 Simulated fedbatch reactor results—used with permission

S_{in}	Influent substrate concentration
$\mu_{\max}$	Maximum specific growth rate
$v_{\max}$	Maximum specific rate of substrate consumption
P_m	Alcohol inhibition factor
Y_p	Substrate-product conversion rate
K_s, K_i, K_{1s}, K_{1i}	Parametric quantities

9.2.2.2 Simulated Results

Figure 9.5 shows typical results [1] derived from this model.

9.2.2.3 Plant Control

In order to control a plant such as this in the conventional way, it is necessary to formulate some measure of performance for any given batch. A batch is considered over when the substrate level, S falls below some predefined minimum and the biomass becomes inactive. Also the feed control (Q) is disabled when the reactor is full. In order to apply Rosenbrock's nonlinear control schema, Quiennec [1] uses the final product yield (Yf) as given in Eq. 9.8 and the cost functional (Jf) as given in Eq. 9.9.

$$Y_f = P_f \cdot V_f \tag{9.8}$$

$$J_f = a \cdot P_f \cdot V_f - b \cdot S_f \cdot V_f - c \cdot T_f \tag{9.9}$$

Equations 9.10 and 9.11 define the terms Yc and Jc as the yield and cost, respectively, at the instant the controller function is discontinued when the reactor is full.

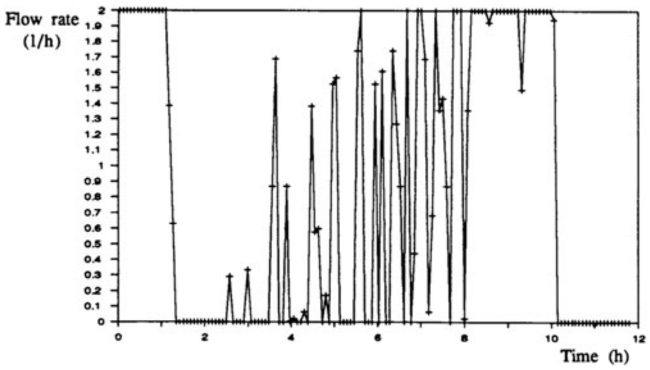


Fig. 9.6 Experimental rig feed rate—used with permission [1]

$$Y_c = P_c \cdot V_c \tag{9.10}$$

$$J_c = a \cdot P_c \cdot V_c - b \cdot S_c \cdot V_c - c \cdot T_c \tag{9.11}$$

Where,

P_f	Final value of product P at time T_f
P_c	Value of product at time T_c
V_f	Final volume of batch V
V_c	Volume at control cut-off (tank full)
S_f	Final value of substrate S
S_c	Value of S at time T_c
T_f	Time at which the batch ends
T_c	Time at which control ends
a,b,c	Weighting coefficients

The process efficiency is defined as the ratio of yield to cost. Using these terms and applying a Rosenbrock's controller an experimental rig showed good agreement with the simulated model. The feed rate (Q) profile that was computed in the experimental rig is shown in Fig. 9.6 and is highly nonlinear. The actuation mechanism for such a duty cycle would be very complex. The next section explains how the same system can be controlled using the BOXES method to control the feed rate in an on/off mode.

9.2.3 Boxes Control of a Fedbatch Fermentor

The major conceptual difference in the control of the bioreactor is that the BOXES method switches the nutrient pump on and off. Referencing Fig. 9.1 the control signal will be connected to the pump's control logic.

9.2.3.1 Selection of State Variables

There are potentially eight state variables, which are:

X	Biomass concentration
$\mathrm{d}X/\mathrm{d}t$	Derivative of biomass—see Eq. 9.1
S	Substrate concentration
$\mathrm{d}S/\mathrm{d}t$	Derivative of substrate concentration—see Eq. 9.2
P	Product concentration
$\mathrm{d}P/\mathrm{d}t$	Derivative of product concentration—see Eq. 9.3
V	Volume of liquid in the reactor
$\mathrm{d}V/\mathrm{d}t$	Derivative of fluid volume—see Eq. 9.4

To make the system somewhat equivalent to the trolley and pole application S, $\mathrm{d}S/\mathrm{d}t$, P and $\mathrm{d}P/\mathrm{d}t$ were chosen as the representative state variables. Of course, the other variables could be included but they would increase the complexity of the algorithm quite considerably.

9.2.3.2 Quantifying the State Variables

The BOXES methodology requires that the state variables be quantified into a unique state number that becomes the key for the table lookup process. In order to make the application truly generic, the state variables as always must be normalized using expected maxima and minima. This process is performed on the selected state variables in order to produces a set of state zone integers in the usual manner. The state integers are combined into a system integer (m) as previously.

9.2.3.3 The Boxes Signature Table

For each value of system integer, the signature table contains a zero or a one corresponding to an on/off signal which is to be applied to the feed pump. The number of items in the signature table is purely a function of the number of state boundaries and is accessed as in all BOXES applications in a read only mode. The frequency of access is either based on the plant's time constant or event driven when the system migrates from one cell into another as shown in Fig. 9.7.

9.2.3.4 Run Time Data

When the system integer changes from one legal value to another, the software records the time of entry into the new cell and updates the performance gain within the last cell. The performance criteria (J) at any instant of time (t) used in Fig. 9.7 is a modified version of Eq. 9.9 which is given below as Eq. 9.12.

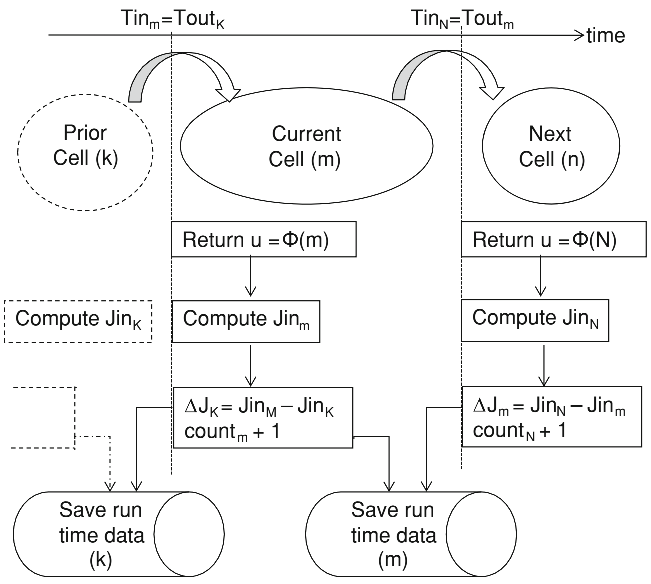


Fig. 9.7 Run time data collection

$$J(t) = a * P_{(t)*} V_{(t)} - b * \cdot S_{(t)} * V_{(t)} - c * t \tag{9.12}$$

Referencing Fig. 9.7, as the bioreactor moves from cell *k* to *m* and then onto *n*, the difference in performance (*ΔJ)* during the time the system spent inside cell is saved along with an incremented count (C_m) of the number of times the cell has been entered. At the end of the run at time T_f the data for the last cell entered is processed using the final values for P_f, *Sf* and T_f in Eq. 9.12.

9.2.3.5 System Merit

In order for the algorithm to learn, the system must be assigned merit values for the immediately prior run and for the global system to date. There are many ways in which these can be calculated based on the designer's skill and experience. One possible algorithm based on cost factors alone is suggested in Eq. 9.13.

$$\text{Merit} = \left(J_f - J_c\right) / \left(J_f + J_c\right) \tag{9.13}$$

Where

J_c	Cost factor when nutrient is switched off $= a \cdot P_c \cdot V_c - b \cdot S_c \cdot V_c - c \cdot T_c$
J_f	Final cost factor $= a \cdot P_f \cdot V_\mathrm{f} - b \cdot S_f \cdot V_f - c \cdot T_f$

9.2.3.6 The Learning Algorithm

A similar algorithm to that used in Eqs. 5.3–5.5 can be adopted to calculate values for global performance and a value for overall system merit and are given as Eqs. 9.14–9.21.

$$\text{global\_perf} = \text{global\_perf} * \delta\text{k} + \text{Jf} \tag{9.14}$$

$$\text{global\_use} = \text{global\_use} * \delta k + 1 \tag{9.15}$$

$$\text{system\_merit} = \text{global\_perf}/\text{global\_use} \tag{9.16}$$

$$\text{dla} = \text{C}_0 + \text{C}_1 * \text{system\_merit} \tag{9.17}$$

The data collected during the last run is now merged with the historical values (JD_1, Co_1, JD_0, and Co_0) stored for each decision.

```
FOR EACH CELL $m$
        IF $\Phi(\text{m}) = 1$ THEN
                $$\text{DJ}_1 m = \text{JD}_1 m * \delta k + \Delta Jm \tag{9.18}$$
                $$\text{Co}_1 m = \text{Co}_1 m * \delta k + Cm \tag{9.19}$$
        ELSE
                $$\text{DJ}_0\text{m} = \text{JD}_0\text{m} * \delta k + \Delta\text{Jm} \tag{9.20}$$
                $$\text{Co}_0 m = \text{Co}_0 m * \delta k + \text{C}m \tag{9.21}$$
        ENDIF
NEXT CELL
```

It is now possible to calculate the strength, or certainty, of the current decision *Φ(m)* for each cell using equations such as Eqs. 9.22 and 9.23 and then alter the signature table value as necessary as shown in Eqs. 9.24 and 9.25.

```
FOR EACH CELL m
        $$\text{On\_strength} = (\text{DJ}_1\text{m} + \text{K} * \text{dla})/(\text{Co}_1\text{m} + \text{K}) \tag{9.22}$$
        $$\text{Off\_Strength} = (\text{DJ}_0 m + \text{K} * \text{dla})/(\text{Co}_0 m + \text{K}) \tag{9.23}$$
        $$\text{IF On\_strength} > \text{Off\_strength THEN } \Phi(m) = 1 \tag{9.24}$$
        $$\text{IF Off\_strength} > \text{On\_strength THEN } \Phi(m) = 0 \tag{9.25}$$
NEXT cell
```

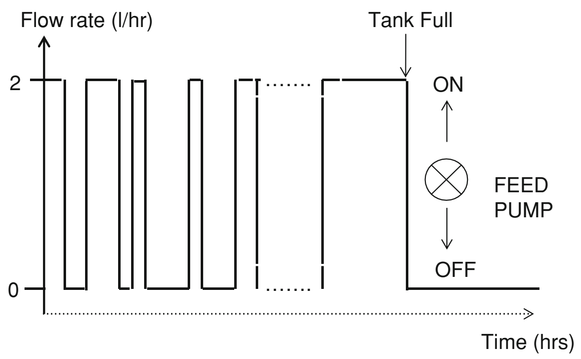


Fig. 9.8 BOXES feed rate profile

9.2.3.7 Results

Figure 9.8 illustrates a typical control profile that is produced over some small time interval near the end of the feeding cycle. It should be noted that this is very simple compared to the pump demands shown in Fig. 9.6. The flow rate would be either fully ON (e.g. 2 l/h) or completely OFF. The pump profile would not be as exact as depicted in Fig. 9.8 as some variance would still occur. The fedbatch system is slow compared to a trolley and pole so the sample rate would be set accordingly.

9.3 A Municipal Incinerator with BOXES Control

A second application uses the BOXES algorithm to regulate when electrical power, a primary cost item in the incineration of municipal waste, is switched on and off in order to maintain the incineration process. This occurs if the incoming trash is wet or partially inflammable. Gasses that are produced as a byproduct of the incineration process are routed through a bag-house containing sacking to remove air borne particles, then through a scrubber in which all soluble compounds are dissolved and the remainder is vented to the atmosphere via the familiar tall chimney. In more modern factories the fumes may be sequestered in underground caverns or bio-farms.

9.3.1 A Model of a Municipal Incinerator

Figure 9.9 is a much simplified schematic of the system. Trash is dumped from municipal collection vehicles into a holding pit. A crane and claw system grabs trash from the pit and drops it into the burning furnace. As necessary, the electrical power system is energized in order to initiate or maintain the incineration process.

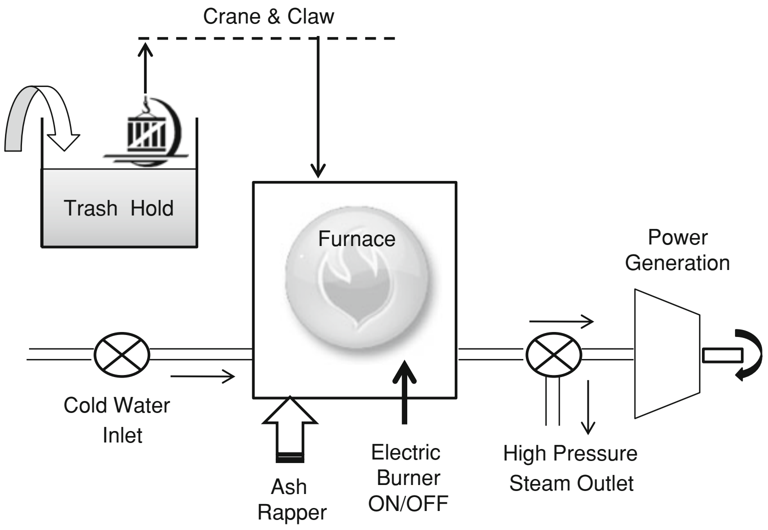


Fig. 9.9 Schematic of municipal plant

When a steady supply of trash is being burned cold water is converted to steam that is used for onsite power generation or office heating. At some preset frequency, the ash rapper system shakes the burning platform free of ash and collects it for disposal in a landfill. Safety features must be implemented to override the intelligent control system. The municipality profits from landfill reduction and the production of super-heated steam.

Russell [7] suggests Eq. 9.26 through 9.30 can be used to simulate such a plant although other models are available.

$$\mathrm{d}B/\mathrm{d}t = K_1 \cdot u + K_2 \cdot F - K_3 \cdot (T - T_0) - K_8 \cdot A \tag{9.26}$$

$$\mathrm{d}A/\mathrm{d}t = K_7 \cdot \mathrm{d}B/\mathrm{d}t \tag{9.27}$$

$$\mathrm{d}F/\mathrm{d}t = -K_4 \cdot \mathrm{d}B/\mathrm{d}t \tag{9.28}$$

$$\mathrm{D}T/\mathrm{d}t = k_5.B - K_6 \cdot (T - T_0) \tag{9.29}$$

$$\text{IF } T > Ts \text{ THEN } S = k_9.B - -K_{10}(T - T_0) \tag{9.30}$$

Where,

B and dB/dt	Energy balance variable and its rate of change
B and dB/dt	Furnace load and its rate of change
T	Water/steam temperature
T_0, T_s	Initial temperature, and boiling point

A	Furnace ash, A is reduced to zero after each rap
S	Steam production
K_1–K_{10}	Model parameters

9.3.2 *BOXES Control of the Municipal Incinerator*

The BOXES methodology can be applied to this system using the on/off control of the electrical power as the control variable (u).

9.3.2.1 State Variables

A suggested set of state variables would be B, $\mathrm{d}B/\mathrm{d}t$ and F. These can be normalized and transformed into a system integer (m) in the usual fashion. The system integer acts as the pointer to the signature table.

9.3.2.2 The BOXES Signature Table

For each value of system integer, the signature table is seeded with zero and one data corresponding to switching the electrical supply ON and OFF. The electrical supply ensures that the furnace remains lit regardless of the condition of the incoming trash or garbage.

9.3.2.3 Runtime and Global Data

In a system such as this, the global and individual runtime data can be calculated based on overall productivity and changes within each cell using changes in furnace load, ash production, and energy used. The ash rapper, which removes any ash build up, will cause the BOXES system to jump between cells that are not necessarily adjacent to the current operating cell as its action is to force the ash value to zero. An inspection of Eq. 9.26 suggests that the burning rate would increase dramatically when the ash term ($-K_8 \cdot A$) becomes zero.

9.3.2.4 Learning Algorithm

The learning algorithm would be almost identical to that for the fedbatch system and based on burning efficiency and volume reduction. The algorithm would evaluate the contribution of each cell and reset the control values in the signature table based on these criteria.

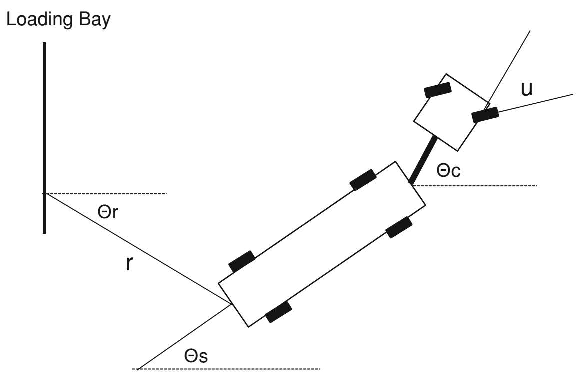


Fig. 9.10 Reversing a tractor trailer

9.4 Reversing a Tractor Trailer

The problem of reversing a tractor trailer has been studied quite extensively and several intelligent methodologies [8, 9] applied. Figure 9.10 illustrates the problem.

The object is to park the trailer so that it is perpendicular to the loading bay by altering the truck wheel angle (u) while in reverse gear.

9.4.1 Model of the Tractor Trailer

The motion of the tractor trailer can be described by the following discrete time model [9] in which a unit time step represents the sampling rate of the state variables.

$$A = p * \cos(\mathrm{u}) \tag{9.31}$$

$$B = A * \cos(\Theta\mathrm{c}(\mathrm{t}) - \Theta\mathrm{s}(\mathrm{t})) \tag{9.32}$$

$$C = A * \sin(\Theta\mathrm{c}(\mathrm{t}) - \Theta\mathrm{s}(\mathrm{t})) \tag{9.33}$$

$$\mathrm{x}(\mathrm{t}+1) = \mathrm{x}(\mathrm{t}) - B * \cos(\Theta\mathrm{s}(\mathrm{t})) \tag{9.34}$$

$$\mathrm{y}(\mathrm{t}+1) = \mathrm{y}(\mathrm{t}) - B * \sin(\Theta\mathrm{s}(\mathrm{t})) \tag{9.35}$$

$$\mathrm{D1} = \mathrm{dc} * \sin(\Theta\mathrm{c}(\mathrm{t})) - p * \cos(\Theta\mathrm{c}(\mathrm{t})).\sin(\mathrm{u}(\mathrm{t}) \tag{9.36}$$

$$\mathrm{D2} = \mathrm{dc} * \cos(\Theta\mathrm{c}(\mathrm{t})) + p * \sin(\Theta\mathrm{c}(\mathrm{t})).\sin(\mathrm{u}(\mathrm{t}) \tag{9.37}$$

$$\Theta\mathrm{c}(\mathrm{t}+1) = \arctan(\mathrm{D1}/\mathrm{D2}) \tag{9.38}$$

$$D3 = ds * \sin(\Theta s(t)) - C * \cos(\Theta s(t)) \tag{9.39}$$

$$D4 = ds * \cos(\Theta s(t)) + C * \sin(\Theta s(t)) \tag{9.40}$$

$$\Theta s(t+1)) = \arctan(D3/D4) \tag{9.41}$$

Where,

x(t), y(t)	Coordinates of the center of the rear of the trailer (m)
Θs(t)	Angle of trailer (°)
Θc(t)	Angle of cab (°)
u(t)	Steering wheel angle (±70°)
p	Distance front tires move in one time step
dc	Length of cab
ds	Length of trailer (m)

9.4.2 BOXES Control of Tractor Trailer Reversal

In a BOXES implementation, the truck wheels are set to be either 70^{o} left or right (full lock) in a bang-bang arrangement.

9.4.2.1 State Variables

For BOXES processing the state variables would be *x(t), y(t), Θs(t)*, and *Θc(t)* which would be divided into zones in the usual manner and combined to give values for a system integer.

9.4.2.2 The BOXES Signature Table

For each value of system integer, the corresponding signature table contains cell values of full lock control values of truck steering wheel angles of ±70^{o}.

9.4.2.3 Runtime and Global Data

The system would record the number and time of entry into each cell.

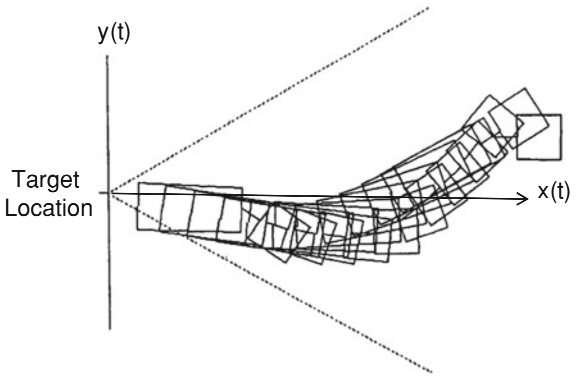


Fig. 9.11 Reversing a tractor trailer 'Courtesy of WIT Press from the book Applications of Artificial Intelligence in Engineering VI—1991—Eds. Rzevski and Adey. Page 914'

9.4.2.4 Learning Algorithm

The global merit would be based on how closely the final resting place of the trailer was or the time to maneuver the trailer into an acceptable position. To effect learning between the cells, each decision would be rewarded or penalized using a performance function such as that given in Eq. 9.42.

$$Jp = (50 - r) * (50 \cdot \cos \Theta - r) \tag{9.42}$$

9.4.2.5 Typical results

Figure 9.11 shows results [9] from a BOXES run for the target location shown.

9.5 Conclusion

In all three case examples given in the chapter, a model of the system under consideration is assumed to be available for simulation and training purposes. It would be customary for simulations of the model to be executed repeatedly until a fairly well trained signature table became apparent. There are many other applications including the operation of a reflux-splitter in a distillation column or a multi-hinged pole [10] driven by a powered cart etc.

Disclaimer
The systems in this chapter appear to promote further research and reference only and should not be construed as being directly applicable to any physical system without further study and testing.

References

1. Quiennec I, Dahhou B & Sevely Y. (1991) Theoretical and Experimental Results in Optimisation of Fedbatch Fermentation Processes. *J. Systems Engineering*, 1(1): 31–40.
2. Constantinides A, Spencer JP. & Gaden EL. (1970) Optimization of Batch Fermentation Processes. II. Optimum Temperature Profiles for Batch Penicillin Fermentations. *Biotechnol Bioeng*. 12: 1081–1098.
3. Calam CT. & Russell DW. (1973) Microbial Aspects of Fermentation Process Development. *J. Appl. Chem. Biotechnol.* 23: 225–237.
4. Williams D, Yousefpour P & Wellington MH. (1986) On-line Adaptive Control of a Fedbatch Fermentation of Saccharomyces Cerevisiae. *J. Biotechnol & Bioengineering*. XXVII:631–645.
5. Rosenbrock, HH., (1960) An Automatic Method for Finding the Greatest or Least Value of a Function. *Compu J*. 3 (3): 175–184.
6. Ghose TK, Tyagi RD (1979).Rapid Ethanol Fermentation of Cellulose Hydrolsate. Product and Substrate Inhibition and Optimization of Fermentor Design. *Biotechnol Bioeng*. 21:1401–1420.
7. Russell DW (1991) Further Studies in AI Augmented Process Control. *Annual Review in Automatic Programming*.Vol:16(1):75–79.
8. Nguyen, DH and Widrow, B (1990) Neural Networks for Self-learning Control Systems. *IEEE Control Systems Magazine*. April: 19–23
9. Woodcock N et al. (1991) Fuzzy BOXES as an Alternative of Neural Networks for Difficult Control Problems. *Applications of Artificial Intelligence in Engineering VI*. Rzevski and Adey RA (eds) CML. 903–919.
10. Wieland AP (1991) Evolving Neural Network Controllers for Unstable Systems. *Proc. IJCNN-91 Seattle International Joint Conference on Neural Networks.* Vol II: 667–673.

Chapter 10
Two Nonlinear Applications

10.1 Introduction

The majority of BOXES research so far has focused on short lived applications such as the trolley and pole, although this book has shown that continuous, long-lived industrial systems may also make use of the useful control knowledge that grows over time in the signature table and its statistics. But there are many other areas of application and two of those will be described in this chapter.

In the first the BOXES method is adapted to assist in the forecasting of the best next records to be accessed in a commercial database system. The BOXES algorithm is connected to the disc access system which itself may be firmware or hardware. The interface between the disc and host processor provides a hook into which the adapted BOXES methodology can be connected. This unit collects disc traffic data and processes it in such a manner that it can learn temporal patterns in the accessing methodology and subsequently offer improved caching strategies.

In the second application the BOXES algorithm is attached to the parameters in the familiar Lorentz [1] equations as an archetype of a chaotic system. Many industrial and laboratory systems exhibit nonlinear transient behavior during their life cycles. Examples include electroplating [2], the Duffing Oscillator [3], and many multiple tank chemical processes. Under certain circumstances this motion is classified as chaotic as the dynamic patterns exhibit the familiar chaotic characteristics of self-similarity and heavy dependence on initial conditions.

10.2 Database Access Forecasting

Written with Dennis Wadsworth—Lockheed Martin Software Engineer—retired

The adaptive BOXES method observes anticipated changes in patterns of disc use in the problem domain and then determines optimized actions for these

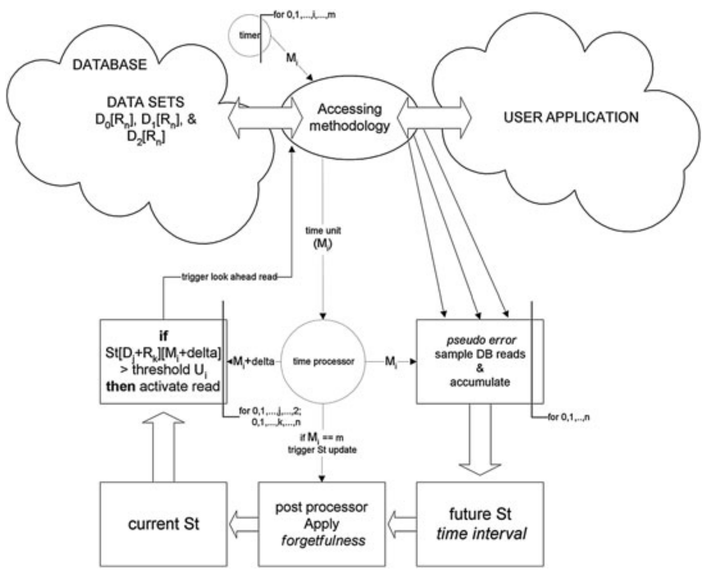

for 0,1,...,i,...,m
timer
M_i
DATABASE
DATA SETS
$D_0[R_n]$, $D_1[R_n]$, & $D_2[R_n]$
Accessing methodology
USER APPLICATION
time unit (M_i)
trigger look ahead read
if St[D_j+R_k][M_i+delta] > threshold U_i then activate read
M_i+delta
time processor
M_i
pseudo error sample DB reads & accumulate
for 0,1,...,j,...,2; 0,1,...,k,...,n
if M_i == m trigger St update
for 0,1,..,n
current St
post processor Apply forgetfulness
future St time interval

Fig. 10.1 BOXES augmentation of database access

projected changes prior to use. This information provides a non-invasive enhancement to the accessing methodology and offers assistance only if its actions are deemed to be appropriate for decreasing access time.

10.2.1 Application of BOXES to Disc Accessing

The adaptive BOXES method breaks the application domain into a finite number of cells that are then evaluated, decomposed, and added to as more knowledge is gained in real-time in a similar manner that digital signal processing (DSP) units analyze data in between data pulses. The algorithm collects data samples at some preset rate and uses them to define a switching surface. The goal is to learn and redefine anticipated requests for data by modifying the switching surface using the previous history of the cells and decrease the learning time.

A decrease in retrieval time is achieved by augmentation of the database's own accessing methodology. The augmentation of the accessing methodology is non-intrusive in nature since the method is contained in the database's own software routines and not in the user application or in the operating system.

Figure 10.1 shows how BOXES is configured to augment the performance of a simple database accessing [4] methodology. The example used in Fig. 10.1 is a simple database that comprises three data sets $\{D_0$, D_1, and $D_2\}$ and uses $\{R_n\}$ as the record index.

The signature table (ST) defines the problem domain and is used to prompt the database to read the forecasted record by updating the internal buffer maintained by the database on a per-structure per-application basis (just) prior to the application's request for the data.

The time-oriented nature of the data is developed as the data-driven user application accesses the database though the AI-enhanced accessing methods over multiple finite time intervals. The pattern reflected in the signature table may be thought of as a switching surface [5] in that it is used to influence future requests.

A point on the surface is described by (M_i, D'_R, U_i), where: D'_R is the unique identifier of the data set $D_{j,}$ connected to a data set record index R_k; and M_i is some unit of time within the finite time interval, and U_i is the value of a statistical inference factor that rates the data set's use at that point of time. The switching surface is nonMarkovian and akin to a rubber sheet in which there are *hills* (maximums) and *valleys* (minimums), and like a rubber sheet the switching surface's maximums and minimums migrate over the life of the database operation.

This threshold can be visualized as a simple plane above the switching surface as seen in Fig. 10.2.

Each cell structure within the signature table holds the statistical data which constitutes the knowledge base of the method. A forgetfulness factor ($\mathrm{ST}[n,t] = \mathrm{ST}[n,t]^{*}\psi$) is applied to the statistical data which serves to refine and smooth the switching surface and to suppress any statistical data skewing inherited from early excursions of the program and the effects of random initial seeding of the control matrix. A typical rule, equivalent to reversing the motor on the familiar trolley and pole system, to trigger the forecasting might be:

```
RULE
  IF a data hill EXCEEDS some preset value < >
  THEN that data is likely for retrieval and cached forward < >
```

10.2.2 Learning in the Adapted BOXES Algorithm

At the end of some finite time interval, the BOXES post-processing routine in real-time recomputes all statistical inference values and updates the Signature Table using a schema of *rewards* or *penalties* as usual. The signature table values represent the probability that a data set will be used during the next time interval. This learning procedure is performed without interfering with normal database traffic.

The prompt to the database *read* directive is triggered when the statistical inference factor within a cell structure is estimated as likely to break a controlled *threshold* (U_i) within plus δ time units. The threshold is described as a simple or complex surface at some height (h) above the switching curve. Delta (δ) is the look-ahead time-factor used by the system to forecast an application's need for

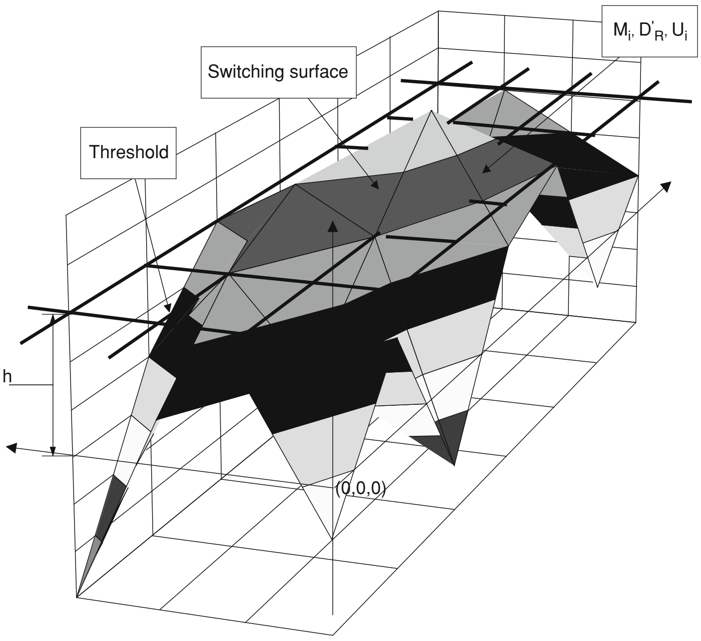


Fig. 10.2 The signature table switching surface with threshold

that data and to issue the corresponding *load cache from record* command. When this occurs, the internal buffer of the database is loaded to cache waiting for the applications probable use. As each time interval is postprocessed the Signature Table cell at (D′$_1$, M$_1$) reflects the change in the statistical inference indicator, U$_1$, over time. The BOXES process updates the switching surface defined by the signature table to achieve the look-ahead goal.

10.2.3 Forecasting

Forecasting is defined by the algorithm given as Eq. 10.1.

$$\begin{aligned}&\text{IF}(U_i = ST[(D_j + R_k)][(\delta + M_i)].\mathit{probability}) >\ = \{\mathit{the}\ \text{``}\mathit{need}\text{''}\ \mathit{rule}\}\\&\text{THEN}\{\mathit{load\ the\ cache\ from\ record}\ (D_j + R_k)\}\end{aligned} \tag{10.1}$$

The signature table (ST) is indexed by some time unit (M_i) plus some look-ahead value (δ), so that the system can test the signature table for possible bring-forward candidates. The look-ahead, or offset factor allows the modified BOXES algorithm to forecast the need (U_i) for any data set record specified within the $\delta + M_i$ time window based on the curve contained [6] in the signature table.

10.2.4 Simulation Results

The database simulation was used to demonstrate and support the non-invasive time forecasting paradigm in a controlled environment. The database accessing methodology is simulated at the record level to chart and implement the adaptive BOXES paradigm. The controlled environment provides the metrics to determine if the system did indeed learn. Figures 10.3 and 10.4 are representative of the results from the simulation and clearly demonstrate the improvement in switching surface produced by the adaptive BOXES enhancement.

The change in the switching surface may be attributed to three factors:

- The maturing of the switching surface as the statistical inference procedure takes effect.
- The application of a *forgetfulness factor* that is applied to each cell structure in the signature table, so that poor, naive, and initial decisions are made progressively less significant over time.
- The different lengths of time interval used in a simulation.

In each simulation the length of the time interval is selected and held constant. The signature table is initially populated with zeroes and data is accrued as real-time progresses. It is apparent from Figs. 10.3 and 10.4 that the longer the activity is sustained, the more defined the statistical peaks become. This indicates that the caching process would proceed with a significant reduction in retrieval time latency.

10.2.5 Conclusion

Any intelligent caching agent involved in a data access methodology is complex with temporal uncertainties that cannot be described by a mathematical model. Adaptive BOXES use a switching surface to influence the accessing methodology through a finite number of statistical inferences. The paradigm's evaluation of just-in-time attempts at caching is used to evaluate its performance resulting in significant economic returns. The method does not require a detailed mathematical model of the system under control and lends itself to this type of non-invasive time forecasting. The objective was to introduce the concept of enhancing the access methodology of a database system by attaching an intelligent agent. The selection of BOXES as that suitable AI agent is due to its statistical and learning capabilities. This section has shown that the signature table maps database activity in such a manner that access predictions decrease retrieval time latency over sustainable finite time periods.

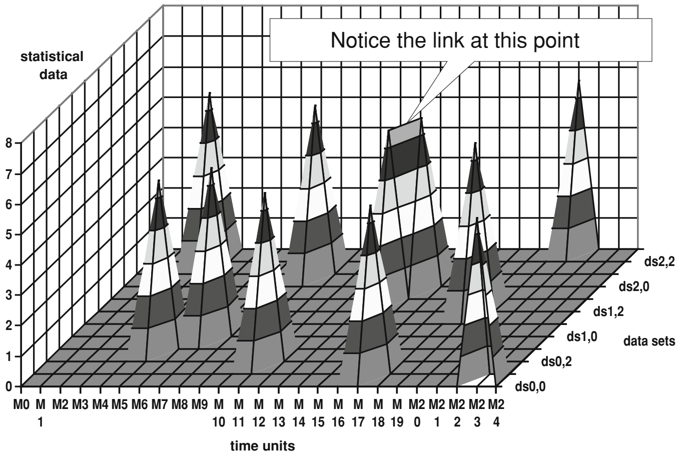


Fig. 10.3 Switching surface early in the BOXES learning process

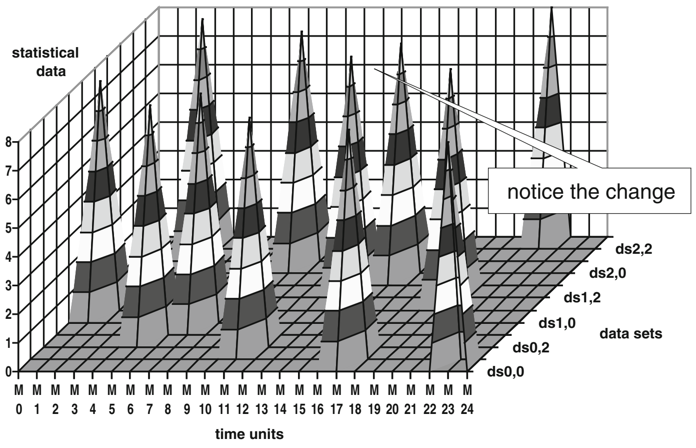


Fig. 10.4 Switching surface with mature BOXES learning

10.3 Stabilizing Lorentz Chaos

Systems, such as the familiar Lorentz equations, offer a real challenge to the control systems because of the chaotic nature of their dynamic behavior that may be sporadic or even random in its onset. A BOXES-type automaton can be constructed [7] to attach to a simulation of the Lorentz Equations and taught to stabilize the motion.

10.3.1 Introduction

There has been a significant increase in interest in chaos and how it relates to real-life systems such as the prediction, and eventual *control*, of weather patterns or the stock market. The element of control is always present and has brought into popularity what has been a well-researched topic for many years. Schelberg [8] uses the New York City taxi-cab system as an example of a chaotic system. The implicit controls for these examples could be the city boundaries, the waiting habits of taxi drivers, and passenger knowledge of locations where cabs are plentiful.

The BOXES method has been applied to several other real systems with encouraging results. Furthermore, the controller learns to cope with data all over its control space and seems independent of the initial state variable conditions. As the system matures, the achievement of a predetermined goal becomes obvious and the performance of the system is positively influenced. It is the goal of this chapter to investigate the ability of the BOXES method to control a system that exhibits transient chaos.

The task for the algorithm will be to reduce the complexity of the chaotic orbits in a simulation of the Lorentz Equations as given by Eqs. 10.2–10.4.

$$\mathrm{dx/dt} = \sigma.(y - x) \tag{10.2}$$

$$\mathrm{dy/dt} = \rho.x - y - x.z \tag{10.3}$$

$$\mathrm{dz/dt} = x.y - \beta.z \tag{10.4}$$

Where,

x, y and z = The state variables.
σ, ρ and β = Parameters related to the physical system

The initial conditions for the state variables are randomly selected and it is noted that common to most chaotic systems, their values do affect the trajectory of the ensuing motion of the system. Watching the pattern develop during the simulation, it is apparent that the system is hunting for solutions; yet while the state variables seem attracted to certain regions, they spin away as soon as they approach the vicinity. However, the system is seen to be bounded, tracing out an endless "saddle" shape. This type of behavior is characteristic of chaotic motion. Figure 10.5 shows a typical Z–X Poincaré [9] map for the Lorentz equations obtained by integration of the equations. The solution spaces, in which the system seemingly could rest with no motion, are called *strange attractors* and are apparent in chaotic systems of this type for example, as shown as the eyes of the butterfly wings of Fig. 10.5.

Moon, [9] in his classic text, introduces *fractal dimension* as a measure of the strangeness of the attractor and which is a numeric descriptor of the complexity of the motion.

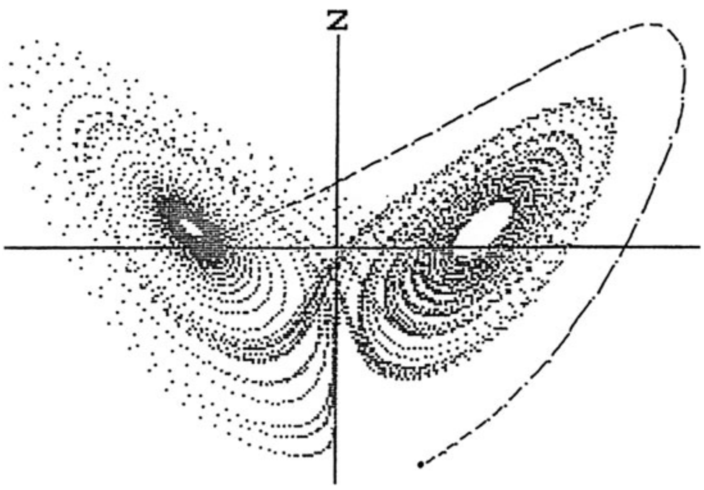


Fig. 10.5 Poincaré map for the Lorentz equations

10.3.1.1 The Information Dimension

There are several types of dimensions [9] that can be calculated from the data that in a Poincaré Map such as the capacity dimension, the information dimension, the point wise dimension, and the correlation dimension. However, the information dimension is similar to some of the algorithms inside the BOXES method. Equations 10.5 and 10.6 detail how this information dimension, called di, is calculated from dynamic chaotic transient data.

$$\mathrm{Pm} = \mathrm{Nm}/N \tag{10.5}$$

$$\mathrm{di} = \lim_{\varepsilon\to\infty} (\Sigma \mathrm{Pm}.\log \mathrm{Pm})/\log(1/\varepsilon)), \tag{10.6}$$

Where ε is a small quantity that is used to divide the state space into *small* cubic cells about which more will be said in the next section. Nm is the number of entries of the system into any given state, *m*; so Pm may be considered the probability of entry into that state if there are total of *N* points generated on the map. If this calculation is performed for example on the shape depicted as Fig. 10.5 above, a non-integer value of 1.764 is obtained for the fractal dimension (di). It is difficult for engineers and scientists to visualize a figure with non-integer dimensions.

10.3.2 BOXES and the Fractal Dimension

The algorithm that implements the BOXES methodology utilizes has several distinct processes that are almost identical to those required in the computation of the fractal information dimension. The state variables are normalized, using known or estimated values of the allowed maximum and minimum for each of the *n* variables.

Each normalized state variable is divided into zones and a state integer calculated at some regular sampled interval of the motion.

10.3.2.1 Quantization

It is necessary to allocate an integer region number for each of the normalized variables that indicate the zone that the state variable is currently occupying. In traditional BOXES systems, each variable has a set of unique boundary values that delineate these zones, but recalling the quantity ε discussed above for fractal calculations, it is easy to imagine that all normalized state variables could be equally divided into 1/ε cells for encoding into state integers.

10.3.2.2 State Number Allocation

The *n* state integers must now be combined into one unique system state number, *m*. It is simple to show by integer arithmetic that Eq. 10.7 performs this function because all values of *r* exist between 1 and 1/ε and all cell zones are of identical size.

$$m = r1 + \sum_{2}^{n} (1/\varepsilon)^{i-1} * (ri - 1) \tag{10.7}$$

This number defines *n*-dimensional space as a sequence of integer states that the *n* variables pass through during their complex chaotic transient motion.

10.3.2.3 The Signature Table

The BOXES methodology uses a signature table (Φ) to store decision values for every value of the state number *m*. Access to any particular decision value, ***u***, is then a simple table look-up process with *m*, as its index (see Eq. 10.8)

$$\boldsymbol{u} = \Phi(m) \tag{10.8}$$

The signature table is initially seeded with a random distribution of decision values and updated after each and every run by a learning algorithm.

10.3.2.4 Attachment to the System

It is proposed that a BOXES-type signature table system can be constructed, so that the entry of the system into a new state triggers the return of a control value as discussed above. In this regard, it is suggested that the information dimension, as defined above, may contain a measure of the overall merit of the chaotic system. Motion with low chaotic components will have a nearly integer value of fractal dimension.

Before such a paradigm can be adopted for the Lorentz system, the connection of the control variable $\boldsymbol{u}$ must be established, numerical values for $\boldsymbol{u}$ determined and learning algorithm constructed. After consideration of the Lorentz equations, it becomes important that the signature table attachment be made to the structure of the chaos, rather than to the system parameters. It is important that the attachment not remove the chaotic nature of the process. It seems a little artificial and trivial to stabilize a chaotic system by rendering it non-chaotic.

The terms in the Lorentz equations that would seem to make the system unstable are the two product terms ($x.z$ and $x.y$) in Eqs. 10.2 and 10.3, respectively. These elements may be considered as virtual driving functions and as such makes them suitable candidates for a BOXES-type attachment. To affect the motion dramatically, it is suggested that Eq. 10.2 be modified to become Eq. 10.9.

$$\mathrm{d}y/\mathrm{d}t = \rho.x - y - \boldsymbol{u}.x.z \tag{10.9}$$

Traditionally, the value of the BOXES variable, $\boldsymbol{u}$, is binary; but in this case it is evident that a more detailed study of the two possible values in the control matrix is required. When the value of $\boldsymbol{u}$ is unity, the system behaves in its original manner, and it is important to design a value for $\boldsymbol{u}$ that maintains system chaos, while enforcing a stabilizing regimen. Studies showed that the other BOXES matrix value, which is now called f, was selected in the range 0.9–0.98, and produced startling reductions in the chaotic phase orbits while preserving the basic dynamics of the system.

10.3.2.5 The Learning Task

An algorithm can be designed to seek to optimize some, usually fuzzy, property or function of the system as a whole using a statistical inference process. BOXES learning algorithms use data acquired during the most recent run and combine it with historical values at the individual cell and global levels. The algorithm produces two factors that represent the worth or strength of both decisions for each cell and preferentially updates the stored value before proceeding to the next run.

10.3.2.6 Learning in the Chaotic Environment

It is apparent that if the data were to be evenly distributed over a few states then the fractal dimension becomes an integer and the more points in each cell the closer the dimension is to zero. The only real measurement of performance available in the Lorentz system is the fractal's information dimension, di. It is proposed that the *merit* of the last run of system be expressed as rdi, the reciprocal of di. This makes it similar to other BOXES automata decision engine methods.

10.3.2.7 Overall System Merit Calculation

When the current run ends, the automaton must incorporate the new data rdi into the historically weighted system merit SYM. Given that: NUR is a measure of the automaton's past experience level and RFD is a weighted accumulated value of all past reciprocal fractal dimensions, then Eqs. 10.10–10.12 show how new values for these global parameters can be computed. The dk term is the usual forgetfulness factor.

$$\mathrm{RFD} = \mathrm{RFD} * \mathrm{dk} + \mathrm{rdi} \tag{10.10}$$

$$\mathrm{NUR} = \mathrm{NUR} * \mathrm{dk} + 1 \tag{10.11}$$

The new system merit is given by:

$$\mathrm{SYM} = \mathrm{RFD}/\mathrm{NUR} \tag{10.12}$$

10.3.2.8 Individual Cell Statistics

After each run, statistics are maintained for each cell that include a historically forgetful accumulation of past runs that are combined with the most recent run. A decision strength is computed for each of the two possible values (1 or f) and the signature table (for that state) updated as appropriate and effects learning.

10.3.3 Results for Lorenz Chaos Under BOXES Control

Figure 10.6 shows a Poincare map illustrating some stabilization of the system shown originally as Fig. 10.5. The computed value for the information dimension in Fig. 10.6 is 0.371 compared to the value of 1.764 in Fig. 10.5.

Figure 10.7 shows a Poincare map illustrating further stabilization of the system as it matures; in this case the value of the fractal dimension di is 0.214. The determined progression toward one attractor and reduced complexity is most interesting.

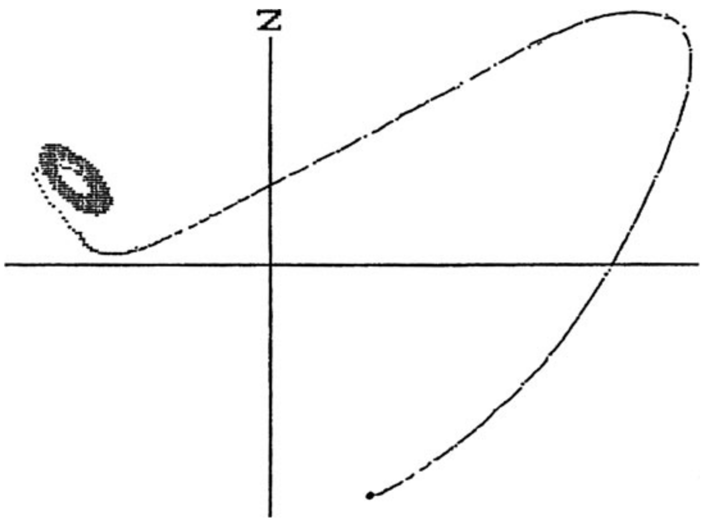


Fig. 10.6 Lorentz equations with BOXES stabilization

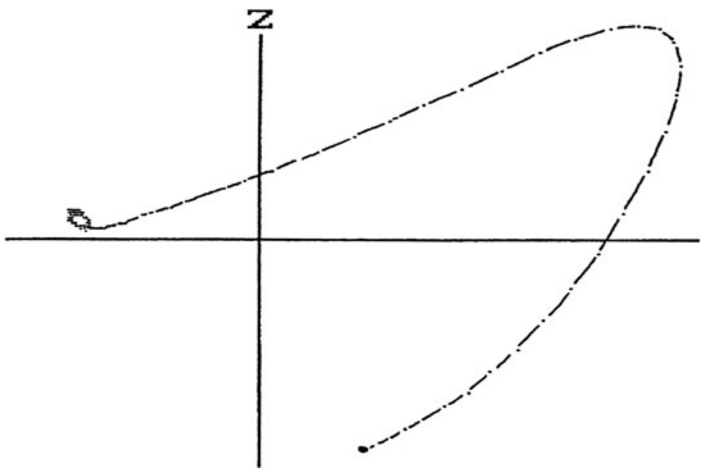


Fig. 10.7 Lorentz equations with mature BOXES stabilization

10.3.4 Conclusions on the Control of Chaotic Systems

Further work is needed to refine the algorithm that computes the linkage between the information contribution and the decision options. The actual connection of the automaton to a real-time, real-world chaotic system is also the subject of much deliberation but the results shown that chaotic tendencies can be suppressed to improve the form factor of the transient.

The dramatic stabilizing effects produced by the attachment of the BOXES algorithm to the well-studied Lorentz equations are most encouraging. Clearly, the calculation of the information dimension requires very similar data capture and processing procedures as those found in the BOXES method.

Other research has focused on adapting the Lorentz parameters but this application of the BOXES method clearly attacks the root of the chaotic terms. The appearance of reported chaotic tendencies in process control [10], medical systems [11], and bio-engineering [12] is noteworthy and may be systems that the methodology could control. This section proposes that a non-iterative, inference method such as the BOXES method may provide a key in the stabilization of such nonlinear systems.

10.4 Conclusion

The applications covered in this chapter exhibit nonlinear dynamic behavior and are on-going areas of study. Large database systems most certainly have repeatable dynamic behaviors and patterns can be established during the day-to-day operations. The BOXES method can provide caching assistance during many transactions. The Lorentz system is a well-known mathematical example that was selected to demonstrate the efficacy of a BOXES attachment in learning to suppress transient nonlinear behavior. Nonlinear and chaotic transients occur in many domains with varying, usually deleterious results ranging from pock marks in an electroplated surface to human seizures. Readers are encouraged to pursue how the BOXES method can implant a modicum of stability in these and other applications.

References

1. Worfolk, P (1995) The Lorenz Equations. Online applet: www.geom.uiuc.edu/~worfolk/apps/Lorenz/ [Accessed July 19, 2011]
2. SG Corcoran and Sieradzki, K. (1992) Chaos during the Growth of an Artificial Pit" *Journal of the Electrochem. Soc*. Vol. 139(6): 1568–73.
3. Novak, F and Frehlich, RG. (1982) Transition to chaos in the Duffing oscillator. *Phys. Rev.* A 26, 3660–66.
4. Russell, DW and Wadsworth DL. (1998) Database Retrieval Enhancement Using The Boxes Methodology, *Artificial Intelligence Real Time Conference (AIRTC)*, October 5–8, Grand Canyon Arizona.

5. Russell DW and Rees SJ. (1975) System Control—a case study of a statistical learning automation, *Progress in Cybernetics and Systems Research* 2: 114–120.
6. Wadsworth, DL (1997) Adaptive BOXES: a learning paradigm. Masters' Thesis. Department of Engineering, Penn State Great Valley, P_A.
7. Russell, DW. (1994) Using the BOXES Methodology as a Possible Stabilizer of Lorentz Chaos. *Proc AI'94 7th.Australian Joint Conference on AI:* Armidale, NSW, Australia 338–345.
8. Schelberg, C (1993) Parallel Scheduling of Random and Chaotic Processes. *Dynamic, Genetic and Chaotic Programming—The Sixth Generation* (ed. Soucek et al.) John Wiley & Sons, New York, Chap. 20.
9. Moon, FC. (1992) *Chaotic and Fractal Dynamics* John Wiley & Sons, New York: 334 ff.
10. D. W. Russell DW (1992) AI Augmented Goal Achievement in Process Control Systems *Proc. 3rd. IFAC Workshop AIRTC'91 Sept 23–25,1991*, Sonoma, CA (eds. M. Rodd & G. Suski) Pergamon Press, New York, pp. 75–79, 1992.
11. Harth, E (1983) Order and Chaos in Neural Systems: An Approach to the Dynamics of Higher Brain Functions" *IEEE Trans. Systems, Man & Cybernetics.* Vol. SMC-13, No. 5.
12. Aihara, K (1991) Chaotic Dynamics in Nerve Membranes and its Modeling with an Artificial Neuron *IEEE Int. Symposium-Circuits*. Vol 3:1457–60.

Chapter 11
Accelerated Learning

11.1 Introduction

There have been several attempts to improve Michie's algorithm. Urbancic and Bratko [1] supplemented the basic BOXES algorithm with production rules in accord with Makarovic's [2] control rules and have attained an increase in learning rate. Recent studies [3] have also shown how the dynamic behavior of the learning algorithm can be positively influenced using selected inclusion of data from neighboring states in the statistical inference procedure.

The original algorithm described in previous chapters is very simple and supports a paradigm of equal opportunity learning by all cells regardless of their locations in the state space. After observing the physical rig, it became apparent that the system was susceptible to poor choices for motor direction in the early stages of learning. To advance learning in the early stages, some preliminary human training was considered to offset the randomness of the initial signature table. However, a human operator did not master the pole balancing task even after many trial runs. Other recent work [4] has, not surprisingly, confirmed that observers must maintain focus on the task and that performance declines most during lapses of attention. Training can improve these abilities and some practice-related improvements did create gains in the ability to avoid mind-wandering, panic, and other lapses in concentration. Because of the poor human performance, it did not seem necessary or viable to allow a human to train the system before allowing it to run in automatic mode. Expert operators obtained average run times (merit) of between 5 and 10 s only while novices failed in under one second. The mechatronic system quickly attained average run times of between 20 and 40 s with or without any human predisposition of data. Simulations attained run times approaching 100 s and showed the possibility of attaining stable limit cycles.

In an attempt to increase the rate of learning in the real and simulated system, the methodology was investigated for possible learning boosts and to reduce any dependency on the initial control values.

First, two fairly simple modifications to the base algorithm are presented in an attempt to accelerate the learning process and cause the initial random values to be

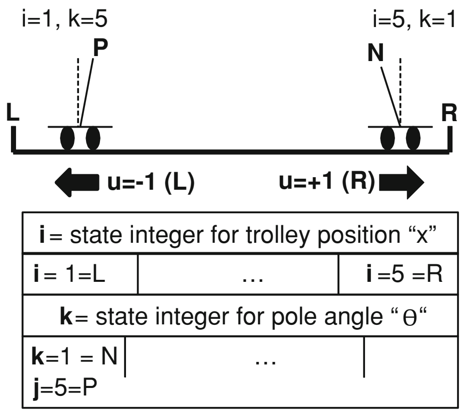

i=1, k=5
P
i=5, k=1
N
L
R
u=-1 (L)
u=+1 (R)
i = state integer for trolley position "x"
i = 1=L
…
i =5 =R
k= state integer for pole angle "θ"
k=1 = N
j=5=P
…

Fig. 11.1 Two situations from which no recovery is possible

quickly forgotten. A second type of modification, which will be expanded upon in Chap. 12 is based on the notion of communal intelligence in which, while the currently active cell remains the primary control agent, other logically adjacent neighbors are allowed to offer advice and potentially reverse the control decision. There are two primary schemas for this which are covered in detail in that chapter. Thirdly, consideration of the internal system structure is made because state integers rely exclusively upon the location of inner state boundaries within the algorithm. The system architect is always left to wonder if these have been placed arbitrarily or optimally, and this is considered in depth in Chap. 13.

11.2 Preset Fixed Signature Table Values

In the specific instance of the trolley and pole system, certain combinations of state integer values prescribe trajectories of the system that seem to require axiomatic motor directions. To illustrate this, assuming that the values of trolley velocity and pole angle velocity are not intuitive to the observer, Fig. 11.1 shows two particular conditions that when they occur will in all likelihood cause the system to fail. If by chance, the motor direction data is as shown, there are up to 18 values of the system integer (m) in the signature table that could promote these seemingly impossible situations. P indicates a positive pole angle and N a negative pole angle.

To identify the 18 possible states that might benefit from fixed values, it is a simple process to create Eq. 11.1 which is identical to Eq. 4.3, with the usual numerical values of state boundaries given in Chap. 4 inserted. Per Fig. 11.1 the state integer for x is "i", for dx/dt is "j", for θ is "k", and for dθ/dt is "l (el)".

$$m = i + 5 * (j - 1) + 15 + (k - 1) + 75 * (l - 1) \qquad (11.1)$$

Table 11.1 States that are candidates for pre-set values

State integer for x is "i"	State integer for dx/dt = "j"	State integer for θ is "k"	State integer for $d\theta/dt$ = "l"	System integer "m"
Left hand end of track ($i = 1$, $k = 5$) $u = \Phi\ (m) = +1$				
$i = 1$	$j = 1$	$k = 5$	$l = 1$	$m = 61$
	$j = 2$		$l = 1$	$m = 66$
	$j = 3$		$l = 1$	$m = 71$
$i = 1$	$j = 1$	$k = 5$	$l = 2$	$m = 136$
	$j = 2$		$l = 2$	$m = 141$
	$j = 3$		$l = 2$	$m = 146$
$i = 1$	$j = 1$	$k = 5$	$l = 3$	$m = 211$
	$j = 2$		$l = 3$	$m = 216$
	$j = 3$		$l = 3$	$m = 221$
Right hand end of track ($i = 5$, $k = 1$) $u = \Phi\ (m) = -1$				
$i = 5$	$j = 1$	$k = 1$	$l = 1$	$m = 5$
	$j = 2$		$l = 1$	$m = 10$
	$j = 3$		$l = 1$	$m = 15$
$i = 5$	$j = 1$	$k = 1$	$l = 2$	$m = 80$
	$j = 2$		$l = 2$	$m = 85$
	$j = 3$		$l = 2$	$m = 90$
$i = 5$	$j = 1$	$k = 1$	$l = 3$	$m = 155$
	$j = 2$		$l = 3$	$m = 160$
	$j = 3$		$l = 3$	$m = 165$

Whenever $i = 1$ and $k = 5$, the motor direction should probably be set to move the trolley to the right ($u = +1$) and conversely when $i = 5$ and $k = 5$, the motor direction should probably be left ($u = -1$). Table 11.1 shows these states. If this is enforced, there will be eighteen less signature table states to be trained, but a new level of software protection will be required to exempt these states from being altered or from contributing as advisors. The initialized signature table will still contain random values of motor direction in all other states.

Considering the trolley only, it may seem that whenever the trolley is nearing one of the extreme state regions (meaning "***i***" is either 1 or 5) that the motor direction should be in the direction that takes it toward the center of the track. However, it should be considered that for the (5 × 3 × 5 × 3) configuration used in previous chapters, there are 45 states that include the values $i = 1$ and 45 that include $i = 5$. This would appear to eliminate 90 states from ever being altered. However, the dynamics of the system allow for the trolley to be set in either direction within those extreme regions because of the other forces exerted on it by the swinging pole. Hence the eighteen states described above would only be expected to make some difference if the system dynamics were never in the situation of small oscillatory motion near the end of the trolley's track, which is a function of how the system is started up. For completeness however, Table 11.2 shows a 225 state integer matrix with these prefixed decisions inserted.

Another problem is that when the algorithm is computing relative strengths (see Fig. 4.10) so that reversals in the signature table data can occur, it would be impossible to assess the strength of a fixed data value.

Table 11.2 State integer matrix showing values of fixed data

Cell	Data values														
01–15					L					L					L
16–30															
31–45															
46–60															
61–75	R					R					R				
76–90					L					L					L
91–105															
106–120															
121–135															
136–150	R					R					R				
151–165					L					L					L
166–180															
181–195															
196–210															
211–225	R					R					R				

Later as discussed in Chap. 13 these fixed state values could dominate those that are computed during real learning runs. As can be clearly seen, the additional overhead would make this learning accelerator much more complex and proved not worth pursuing.

11.3 Reduction of the Importance of Short Runs

A second, more equitable method [5] was to modify the BOXES algorithm so that it could downplay the significance of short non-productive runs. This corresponds to the mulligan in golf in which a player is allowed to retake an errant tee shot with no penalty.

Consider a state that is currently asserting a LEFT control value. The original algorithm computes the certainty or strength of the decision using Eqs. 5.10, 5.12, 5.14, and 5.15 which are repeated here for clarity as Eqs. 11.2–11.5.

$$\text{left\_count}_i = \text{left\_count}_i{}^* \, \delta k + n_i \tag{11.2}$$

$$\text{left\_life}_i = \text{left\_life}_i{}^* \, \delta k + \text{tim}_i \tag{11.3}$$

$$\text{dla} = C_0 + C_1{}^* \, \text{merit} \tag{11.4}$$

$$\text{left\_strength}_i = \frac{(\text{left\_life}_i + K^* \text{dla})}{(\text{left\_count}_i + K)} \tag{11.5}$$

Where,

n_i	Number of times this cell was entered during the previous run
tim_i	Time spent inside the cell during the previous run
C_0	Learning constant coefficient

C_1	Learning rate coefficient
dla	Desired level of achievement
K	Desired learning rate

In some versions of the software, the time component (*tim*$_i$) was calculated using Eq. 11.6. Rather than measuring the total time that the system dwells inside any given cell, this method simply keeps a running total of the time at which the system entered the cell and relates to the overall survival time of the system.

$$\mathrm{tim}_i = \sum_{1}^{n_i} (\mathrm{Tf} - T_i) \tag{11.6}$$

Where,

Tf	Time to failure of the previous run
T_i	Sum of all entry times into the cell during the previous run

The desired level of achievement given by Eq. 11.4 is linearly dependent on the current value of overall global merit which approximates to the average control run time to failure over many control excursions. It is apparent that a poor control run caused, for example, by untrained control decisions can permanently skew the statistics and prevent learning. To compensate for this, the value of *tim*$_i$ was modified using Eq. 11.7.

$$\mathrm{tim}_i = \sum_{1}^{n_i} (\mathrm{Tf} - T_i) + Q * (\mathrm{Tf} - \mathrm{merit}) \tag{11.7}$$

where,

Q	a constant weighting factor always greater than zero

Figure 11.2 illustrates the effect of this modification on merit over time.

Naturally, the optimal value for Q would be difficult to establish and require much testing and experimental design.

11.4 Collaboration Among Cells

So far, the control knowledge that generates a decision has been almost exclusively vested in individual cells which correspond to board positions. Appropriate action is performed by identifying the current state integer or scenario and reading a value from the signature table. This is true for the genre of single action (or player) games, but many games or systems rely on the skill of other players (or tutors) who contribute to the continuation of the game. As learning proceeds, it may prove beneficial for individual cells to consider advice from other tokens or players. In the control system application this makes even more sense because the boundaries that surround any one cell are arbitrary, so the next cell may well have important knowledge to impart to supplement an impending decision. Peer to peer learning occurs in the classroom during group activities or projects.

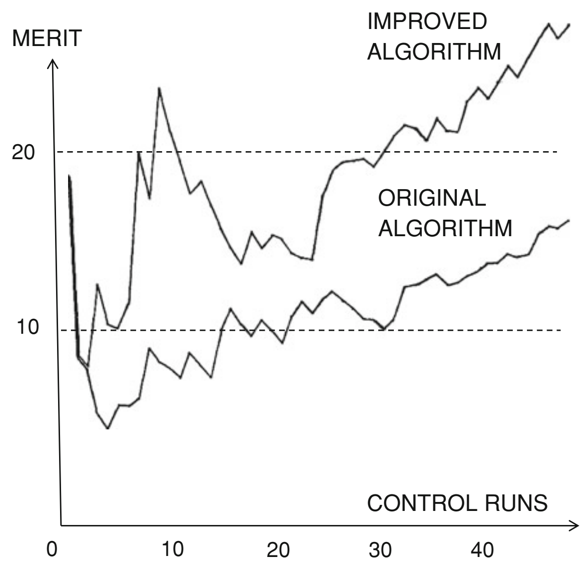


Fig. 11.2 Effect of the improved algorithm on merit

11.4.1 Playing Bridge

Perhaps the most well-known collaborative game is bridge. Players, while not cognizant of the cards in their partner's hands, converse with each other using a bidding process in which each player covertly declares the strength of their hand of cards. Once a contract is decided upon, the players lay down their cards in turn and try to complete their contracted plan. The skill of each player increases with experience and as partners learn the codes in which the game is shrouded. A brilliant player is only as good as the partnerships that are formed with other players. The game is bounded as there are only 52 cards, yet it is very complicated because of player-partner combinatorial sequences.

As will become important later, the partners win or lose together and expertise and experience are shared commodities. In a bridge club, each player is assigned different partners from time to time, which increases the complexity of play, but serves to expand the horizons of all. The ideology is that there is benefit in the players not having complete autonomous control of their shared destiny.

11.4.2 Swarm and Ant Colony Systems

Taking collaboration to the other extreme, a popular control and scheduling mechanism has been found effective that uses the particle swarm or ant colony metaphor. In this scenario, a model is constructed that uses the mechanisms that are used by swarms of bees, bats, or colonies of ants as they forage for food to produce solutions to hard or otherwise intractable problems especially those that involve multiple options and many branched decision trees. Wikipedia [6] defines the ant colony metaphor in layman's terms as:

> … ants apparently wander randomly, and upon finding food return to their colony while laying down pheromone trails. If other ants find the trail they are likely to give up their random paths and follow the trail, returning and reinforcing it if they eventually find food… Ant colony optimization (ACO) algorithms that mimic this behavior have been used to produce near-optimal solutions to problems such as the traveling salesman problem. They have an advantage over simulated annealing and other genetic algorithm approaches in that when the path changes dynamically, the ant colony algorithm adapts to (those) changes in real time.

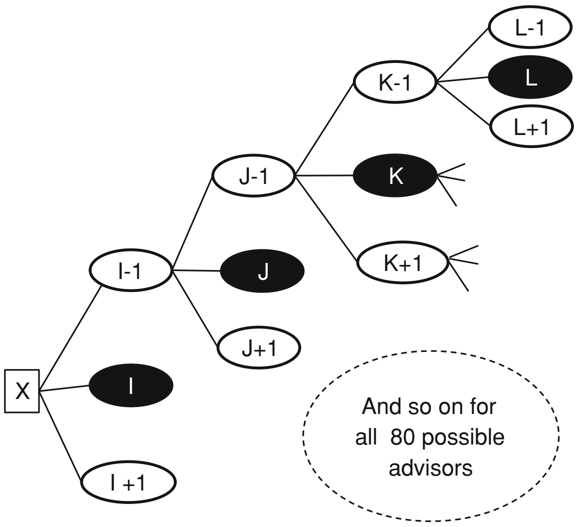
An application of this technology to shop floor scheduling [7] in a hybrid flow shop has been reported. This particular application has been shown to be NP-hard because it involves the dynamic assignment of many, similar, parallel, machines into configurations that satisfy complicated manufacturing or assembly requirements. In this situation, each machine agent is considered an ant in the colony constituted by the shop floor. What is significant is that while each machine is an important contributor to the overall schedule, it is the communal knowledge of all members that can satisfy many of the constraints and implement a viable schedule.

11.4.3 Advisors in the BOXES Algorithm

The positioning of individual sectors of the state variables is somewhat arbitrary and so it can be conjectured that immediately adjacent, sometimes called nearest neighbor cells, might have statistical and control value [8] in helping selected (natural) cells in operating the system. If the cumulative value of the advice outweighs the natural strength of the current cell, the advisor's aggregate decision (to be described in detail in Chap. 12) should prevail and a modified control value (u) suggested by the advisors which may mean ignoring the cell's own natural value. If the advice is considered weak or inconclusive, the natural stored value is asserted and no further action taken.

What is quite unique in this algorithmic change to the BOXES method is that during the post-processing phase, the advisor cells are also statistically rewarded or penalized based on their contribution or lack of contribution. The peer to peer advisors contribute intelligence to the single cell because all cells have varying strengths and experience levels. Because the advisor cells may themselves not be well trained, failure in advising can be programed into their learning processes. This implies that even if a cell does not contribute directly to a decision its impact is still considered and its data re-evaluated. This incidental learning is equivalent to that gained by bystanders or observers.

Fig. 11.3 Location of all possible advisor cells



11.4.4 Locating Peer Advisor States

The location of advisor states might appear quite simple at first, but this is not so. Consider the configuration used in Sect. 11.2 using states near the middle of each of the state variables. If a cell is close to a system boundary then the number of branches in this tree is obviously much smaller. It is apparent that because the total number of possible state integers for this configuration is 225, there are up to 80 possible advisors for each system integer. This indicates that 35% of the signature table cells could influence every cell during the real time operation of the system. This is untenable and illogical in such a state-based methodology. Figure 11.3 shows how there can be up to 80 advisors for each cell.

If complete advisor trees are allowed, the paradigm would become more swarm oriented, and is not considered feasible for a real-time BOXES implementation. In the spirit of near neighbor advising, the number of advisors can be reduced by considering only those cells that differ by one integer position from each of the natural state integers taken one at a time. Figure 11.4 illustrates this and shows that the maximum number of advisors is now eight, which is much more manageable in real-time. Again, if any state integer is near a boundary, then the out of bounds advisor cell is discarded. The system integer variable (m) is unpacked to give its four component state integers i, j, k, and l which are then varied by plus and minus one in turn to create the near neighbor advisor space.

In either of the two cases mentioned above, the four advisor state integers are recombined into an advisor system integer and the decision value and any other data pertaining to that cell is accessed as for any cell identifier.

For example, if the current system integer (m) is 141, Table 11.2 indicates that its four (natural) state integers are $\{i = 1, j = 2, k = 5, \text{ and } l = 2\}$, then Table 11.3,

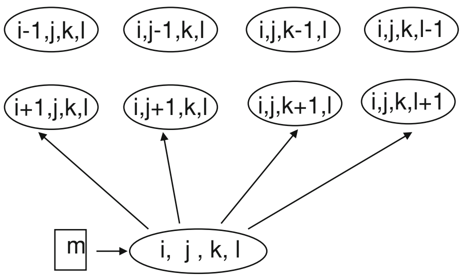


Fig. 11.4 Near neighbor advisor cells

Table 11.3 Example of near neighboring states

i value	j value	k value	l value	Advisor cell #
$i-1=0$	$j=2$	$k=5$	$l=2$	Ignore $i<1$
$i+1=2$	$j=2$	$k=5$	$l=2$	142
$i=1$	$j-1=1$	$k=5$	$l=2$	136
$i=1$	$j+1=3$	$k=5$	$l=2$	146
$i=1$	$j=2$	$k-1=4$	$l=2$	126
$i=1$	$j=2$	$k+1=6$	$l=2$	Ignore $k>5$
$i=1$	$j=2$	$k=5$	$l-1=1$	66
$i=1$	$j=2$	$k=5$	$l+1=3$	216

using Eq. 5.7 for the given {5,3,5,3} configuration [9] shows how the six near neighboring states {142, 136, 46, 126, 66, 216} would be selected. It should be noted that two potential advisors lay outside of the problem space, one at each end. It is clear that there is no linear sequence. Whenever the system integer (m) is calculated, from the normalized state variables, as 141, the information in cells 142, 136, 46, 126, 66, and 216 is retrieved and used to qualify the natural data in cell 141.

The premise is that these closely related cells might be more experienced than the natural cell and offer valuable insight into better control actions. The BOXES algorithm can be modified to include such actions. What was found interesting [10] is that it seemed only fair to reward and punish the peer advisors based on any identifiable resulting change to the values of merit.

If poor advice was given resulting in a cell to falter in its resolve to return its natural control value, then those detractor cells should be weakened so as to be eventually more moderate in their advice. This modification will be covered extensively in Chap. 12.

11.5 Relocation of Boundaries

One of the primary advantages of the BOXES method is its independence from almost all a priori system data. It would appear that when state integers are being formed (for example, see Fig. 4.2) the location of the sub dividers inside any normalized state variable is quite arbitrary. This was considered in some depth in the

Liverpool studies which concluded that the value of the boundaries was only somewhat consequential in the control process. However, the number of boundaries might be more promising as a variation on the system design. The problem with increasing the number of possible state integers for any state variable is the substantial increase in the total number of states to be handled. To illustrate this, if each of the state variables was divided into $5 \times 5 \times 5 \times 5$ instead of $5 \times 3 \times 5 \times 3$ the total number of BOXES cells would increase by 270% to become 625. This aspect of BOXES system design will be discussed in much more detail in Chap. 13 where fuzzy box [11] membership and evolutionary algorithms are also considered.

11.6 Conclusion

The BOXES algorithm can certainly be modified to accelerate learning by adjusting logical and learning components and configurations inside the algorithm and is the subject of study by many researchers across the world. Sammut [12] suggests several variations that were studied at the University of New South Wales that appear very feasible and worth pursuing. Each suggestion achieves some degree of learning improvement usually at the cost of added computational complexity. For example, Cribb [13] introduced a variation to the BOXES method in which each box used a local merit function and argued that the level of exploration within any box should be tied to the performance of the box rather than the whole system. Thus, he devised the notion of local merit, replacing global merit. Local merit was found to halve the number of trials to learn to control the pole and cart. However, it can be argued that it is not really important how long the system dwells in each state rather than the system learns to seamlessly transition between states based on a defined switching surface. If each box is individually important the number and position of state boundaries become a significant system design requirement. These and other alterations can be included in any BOXES software design but can prove problematic from a real time standpoint or may prove inept until the system gains enough maturity to be trustworthy.

The chapter has described four possible system modifications: fixing the decision in some cells; avoiding the impact of short runs; collaborating cells; and relocating state boundaries. Each is summarized below.

A problem with using the fixed cell modification is that because all four state variables contribute to the value of system integer [14] it was frequently observed that sometimes the trolley would oscillate back and forth over a small region close to the end of the track. This was due to the dynamic influence of the pole inertia and a tendency for the system to jitter about a state boundary. This implies that just because the trolley is in an extreme position, it is not always true that the end cell should drive the system in a direction away from the end of the track. A possible more extensive software change over rewarded the system every time the trolley crossed the center of the track, or when the pole swung across the vertical position. This adds a dimension of preferred operational system space which negates the concept of a black-box controller.

The idea of downplaying short runs in the beginning stages of learning is quite effective but violates the usefulness of early stage state space exploration which may lead to preferred patterns that are equivalent to local minima and maxima. The whole state space should be explored for completeness.

The usefulness of advisor states was introduced and will be pursued further in Chap. 12. The black-box signature table control paradigm can be extended to reduce its dependency on the single decision contained in any unique cell using near neighbor, peer to peer, advisors. Care must be taken to limit the number and depth of advice that advisors can provide to avoid a swarm mentality to overarch the BOXES logic. Advice can be given in two ways. The first is to enact a simple voting mechanism in which, regardless of the experience of the advisor, signature table values, which are plus and minus one, for the natural cell and its advisors are arithmetically summed and the sign of the sum returned as the control decision. Another advisor-based implementation is to use the experience and strength of the natural cell and weigh it against the advice from the advisors before any alteration is considered.

The location of boundaries is a very complex matter and will be the topic of Chap. 13 and provides a possible evolutionary method of establishing the system switching surface without any mathematical treatment of the system equations.

In summary, the BOXES algorithm can be written with many variations and has been the basis of much research in learning systems and genetic algorithms.

References

1. Urbancic, T. and Bratko, I. (1992) Knowledge Acquisition for Dynamic System Control. *Dynamic, Genetic and Chaotic Programming*, ed. Branko Soucek and the IRIS Group, John Wiley & Sons Inc. New York, pp 65–82
2. Makarovic, A (1991) A Qualitative Way of Solving the Pole Balancing Problem. *Machine Intelligence 12* ed. Hayes, J, Michie, D and Tyugu, E. Oxford University Press, pp 241–258.
3. Russell, D W (1993) SUPER-BOXES: An Accountable Advisor System. *Proc. 8th Int Conference on Applications of AI in Engineering, AIENG'93*, Toulouse, France. Elsevier Applied Science, London, pp. 477–489
4. Carp, J and Park, J. 2011. Human Learning Improves Machine Learning: Neural and Computational Mechanisms of Perceptual Training. (2011) *J of Neuroscience,* 31*(11): 3937–3938*
5. Rees, S.J. PhD Thesis. 1977. Liverpool Polytechnic
6. Ant Colony Optimization. Wikipedia. The Free Encyclopedia.http://en.wikipedia.org/wiki/Ant_colony_optimization (accessed May 17, 2011).
7. Alakyran et al. 2007. Using ant colony optimization to solve hybrid flow shop scheduling problems. *Int. J. of Adv. Manufacturing Technology* 35:541–550.
8. Russell, D.W., 1995. Advisor Logic. *Control Engineering Practice* 3(7): 977–984 July, Oxford: Pergammon Press.
9. Russell, D.W., Rees S.J. and Boyes, J.A. 1977. A Micro-system for Control by Automata in Real-Life Situations. *Proc. Conf. on Information Sciences and Systems.* Johns Hopkins University, Baltimore. 226–230
10. Russell, D.W. An Advisor-enhanced Signature Table Controller for Real-time, Ill-defined Systems. *Proc. IFAC Symposium on Real Time Control SICICA'94*. Budapest, 1994 66–71.

11. Woodcock N. et al. Fuzzy BOXES as an Alternative to Neural Networks for Difficult Control Problems. 1991. *Applications of Artificial Intelligence in Engineering VI.* Computational Mechanics Institute, Elsevier Applied Science. 903–919.
12. Sammut C. 1995. Recent progress with BOXES. *Machine Intelligence 13* Oxford University Press, Inc. New York, NY, USA
13. Cribb, J. (1989). Comparison and Analysis of Algorithms for Reinforcement Learning. Honours Thesis, Department of Computer Science, University of New South Wales.
14. Russell, D.W. and Rees, S.J. 1975. System Control—a Case Study of a Statistical Learning Automaton. *Progress in Cybernetics Systems Research* 2:114–120, New York: Hemisphere Publishing Co.

Chapter 12
Two Advising Paradigms

12.1 Introduction

Machine learning can be accomplished using algorithms that are designed to search out solutions by dividing the overall problem into sets of local subproblems. The learning with a critic paradigm tutors on an individual state by state basis and leans heavily on the belief that the overall purpose gaining system knowledge is attained by exercising individual cells. In process control, the BOXES method reads control decisions from a memory resident signature table structure using a single system integer as an index. This chapter expands upon the notion that the BOXES black-box signature table control paradigm can be extended to reduce the dependency on the single decision contained in any unique state using the information contained in near neighbour advisor cells as mentioned in Sect. 11.4.

Advisor logic can be designed in such a way as to modify control decisions in several ways and at various times. One method is to let the selected advisor cells vote-based using their left/right control decisions as they appear in the signature table. Another method is to weigh the importance of the advice offered by the advisor cells using their decision strengths. Advisors may contribute useful additional intelligence to the single cell value as a peer group. Of course, the advisors may be as untrained as the natural cell. After each control sequence, the advisors are rewarded or punished based on how the system performed during the last run.

Because the method by which each state variable is converted into a state integer uses arbitrarily placed zone boundaries, the notion of advisor cells may enable the system to take advantage of other cell structures within the problem domain. This process alleviates the physical uncertainty associated with the location of cell boundaries, which may be a major flaw in the BOXES paradigm. In a bang–bang control system such as in the forced trolley and pole system considered throughout the book, it is to be expected that certain popular zones or groups of zones will appear over time. As the system matures, most of the data set in the signature table will contain valuable information about the controllability of the system and possibly where control surfaces are located.

The switching surface has been shown [1] to lie between clearly defined bi-polar regions. A Pontriagin [2] analysis of the system model confirms that optimal control is achievable using a *bang–bang* schema, which means that the optimal solution is contained inside a sequence of binary (LEFT or RIGHT) motor control decisions within the state space. The control task simply translates to providing the motorized cart with sequences of these LEFT and RIGHT switching commands. The system is deemed to fail if any state variable falls outside of its own preset boundaries, in which case, the BOXES algorithm halts and authorizes an update of its signature table control matrix and restarts the system with random initial conditions as explained in Chap. 7.

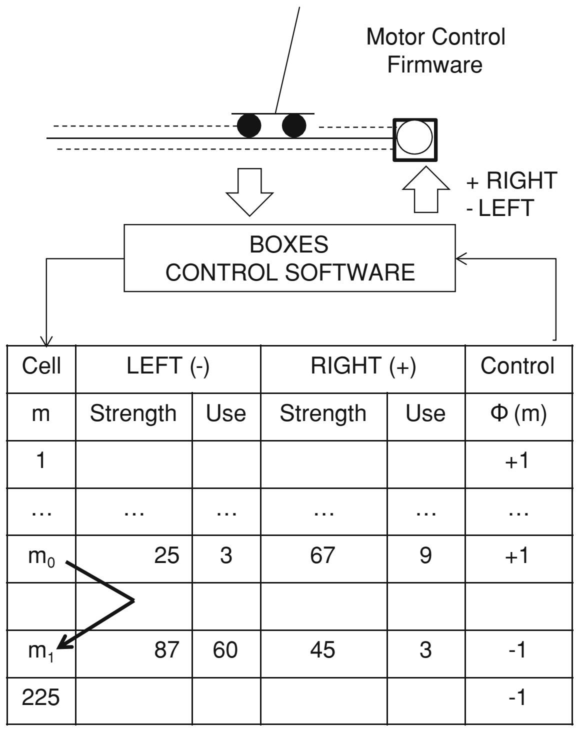
In order to evaluate the efficacy of control, it is commonplace to monitor the progression of system merit over many training runs. The merit of the system, for the trolley and pole, is based on an average value of controlled runtimes. The learning objective is to increase this global merit value. Statistical data are all weighted by a system forgetfulness factor in order to reduce the significance of early experience and eliminate inadvertent skewing due to the initial random control matrix. As the system learns to perform the task of balancing the pole, if the value of system merit increases those cells that contributed to the success are rewarded. Conversely, if the last run produced a shorter than average performance, all cells that contributed to the poor performance must be penalized.

During a control run the state variables alter in real-time and the corresponding state integers vary accordingly. The change in system integer causes a new value of control value to be read from the signature table which may or may not be the same as that currently in force. In order for peer advising to be possible, the natural cell must first identify and select potential advisors, using the principle of nearest neighbor classifiers [3] and then rank them according to positional phyla as it is entered. It is obvious that there will always exist a set of juxtaposed states that vary by plus or minus one zone in any combination of the state quanta. Because a selected advisor state may not be legal because it falls outside of its allowable range, the number of advisors will vary according to the operational domain of the system at any time. Before allowing advice from any advisor cell to be considered, it is important that it be examined for strength and experience.

12.1.1 Decision Strengths of Cells

The location and number of boundaries for each of the state variables attached to the BOXES system is of paramount importance. Russell [4] in a critique of the BOXES method observes that these boundary values present a potential flaw in the methodology. Using the trolley and pole as an application, the BOXES method uses these boundaries to produce four state variables to define the system. As explained in previous chapters (for example, see Sect. 5.2.5) the four variables are coded into four operating zone integers which are combined into an integer which is designated m_0 in Fig. 12.1 and represents some hypothetical current value of the system's state.

Fig. 12.1 BOXES method showing decision strengths



Cell	LEFT (-)		RIGHT (+)		Control
m	Strength	Use	Strength	Use	Φ (m)
1					+1
…	…	…	…	…	…
m_0	25	3	67	9	+1
m_1	87	60	45	3	-1
225					-1

If one of the state variables drifts into a different legal zone by crossing at least one boundary, a completely new system integer (m_1) must be computed. In the BOXES algorithm this returns a new signature table value which may or may not call for the motor to be reversed regardless of how strong the new cell is. Assuredly m_0 and m_1 are neighbors, but because the four dimensional system integer vector is not linear or easily visualized they may be located almost anywhere in the signature table. Figure 12.1 illustrates this and expands upon the concept of decision strengths which was covered in Sect. 5.5.1.

Figure 12.1 shows that during its last update the reason that the control decision for cell m_0 is plus 1 (RIGHT) was because its right decision and experience (use) data values exceeded those for a left decision. Conversely, when the system moves to an adjoining state (m_1) the left decision was stronger. If the illustrated transition occurred the trolley motor would be reversed. The decision strengths are computed from the history of each state and the current merit and current desired level of achievement as repeated here for clarity as Eqs. 12.1–12.3.

$$\mathrm{dla} = C_0 + C_1 * \mathrm{merit} \tag{12.1}$$

$$\mathrm{LS}_i = \frac{\{L\mathrm{life_i} + \mathrm{K} * \mathrm{dla}\}}{\{L\mathrm{use}_i + K\}} \tag{12.2}$$

$$\mathrm{RS}_i = \frac{\{R\mathrm{life_i} + \mathrm{K} * \mathrm{dla}\}}{\{R\mathrm{use}_i + K\}} \tag{12.3}$$

where, LS_i LEFT strength for cell i; RS_i RIGHT strength for cell i; $L\mathrm{life}_i$ LEFT history in cell i; $L\mathrm{use}_i$ Number of L entries into cell i; $R\mathrm{life}_i$ RIGHT history in cell i; $R\mathrm{use}_i$ Number of R entries into cell i; C_0 Learning coefficient; dla Desired level of achievement; C_1 Learning coefficient; K Some desired learning rate.

In the original algorithm, the values of left and right strength are compared during each update interlude and the control value in the signature table adjusted accordingly regardless of the location of the cell in the system. Naturally cells that are entered frequently become more experienced as indicated by their *use* data value while those that have longer times (*life*) spent within them are more efficacious. In applications other than the trolley and pole, cell strengths can be calculated by dividing their life data by their use to give an average value that is independent of the system's overall performance.

12.1.2 Identification and Ranking of Advisor Cells

In the BOXES method, a set of zone integers $\{i_1 \ldots i_n\}$ defines the system at any time within a given sample period. If the number of boundaries for each variable is contained in the set $\{\mathrm{Nb}_1 \ldots \mathrm{Nb}_n\}$, simple integer arithmetic yields a unique state number (m) for any given set of zones per Eq. 12.4 which appeared also as Eq. 5.7.

$$m = i_1 + \sum \left(\prod_{i=2}^{N} \prod_{k=2}^{i-1} (\mathrm{Nb}_i) \cdot (i_k - 1) \right) \tag{12.4}$$

Because of this, it is possible algorithmically to unpack any system integer into its corresponding state integers. Appendix **B** contains an outline of an algorithm that performs this. Knowing the component state integers, as introduced in Sect. 9.4.4, there are several ways in which advisor cells can be selected. Russell [5] introduced the notion of strong and weak advisor cells based purely on their relative location within the state space to the currently active cell. Figure 12.2 illustrates this concept for two state integers. The strong cells are those that vary by only one zone from the natural cell and are the more likely to contain valuable data.

Using a simple rubric such as this, it is a simple matter to rank each advisor cell based on its proximity to the current natural cell during the advisor selection process. Using this it is possible to rank advisor cells by counting how many state boundaries would need to be crossed in order for the advisor state to become the next operational state regardless of how many state integers there are. If there are N state variables, then the ranking would be from 1 to N. The advisor algorithm assigns weighting factors that are inversely proportional to this positional ranking which may be inserted to provide the selected cell's voting strength. In this way, the direct proximity of the advisor to the natural cell affects its voting power. For

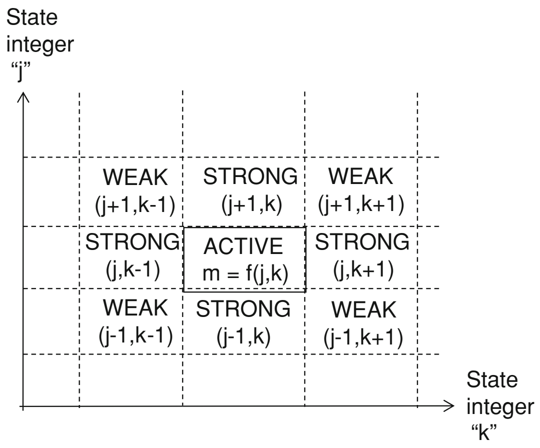


Fig. 12.2 Weak and strong advisor locations

example, a strong cell may be allocated four votes while a weak cell may be limited to only one.

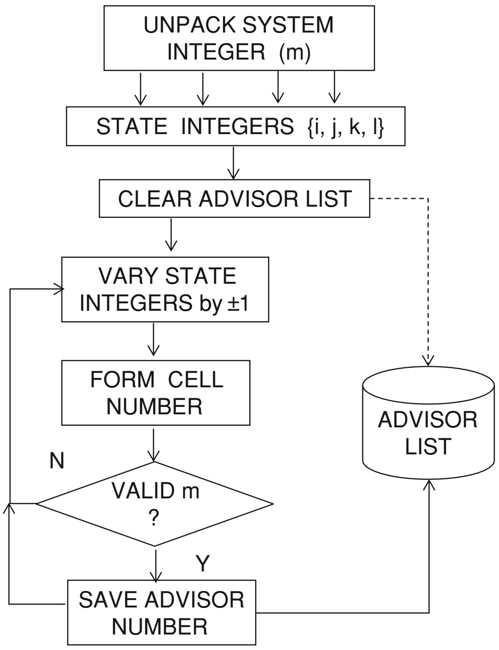
With four state variables the maximum number of advisor states if every combination of adjacent cell numbers is used for each cell is 80. While this is always possible, it is more likely that when the system passes into another state, only one of the state integers will change and then by one zone. Extending Fig. 9.2 to four integers suggests that there are really only eight near neighbours which may prove sufficient for the advising process. If an advisor cell number is valid, the number of advisors for the active cell is incremented and the advisor number saved. If a generated advisor cell falls outside of the system space it is simply ignored and not included in the list. While the advisor list includes a count of how many legal advisors there are, regardless of strength they can be tagged as being strong or weak. A running count of how many times the advisor is called upon to act must be kept in order that accountability can be imputed during the update phase of operation.

Figure 12.3 shows how lists of possible advisors are created for any system.

12.1.3 Advisor Strength Criteria

As Fig. 12.2 indicates, selected advisor cells can be classified purely by their relative position to the active cell. But because each advisor cell exists already in the signature table, it possesses its own set of data items which contain information that is used by the advisor agent in determining their relative strengths. If a cell is experienced and has been entered many times and returned a control value that increased the system merit, it is a strong cell.

Figure 12.4 is an example of the data that an advisor agent might need in order to be effective in the decision process. Figure 12.4 introduces two new terms; a



UNPACK SYSTEM INTEGER (m)
STATE INTEGERS {i, j, k, l}
CLEAR ADVISOR LIST
VARY STATE INTEGERS by ±1
FORM CELL NUMBER
N
VALID m ?
Y
SAVE ADVISOR NUMBER
ADVISOR LIST

Fig. 12.3 Generation of advisor cell lists

Fig. 12.4 Advisor strength terms

For each BOXES cell (e.g. 1..225) with advisors				
	For each advisor (e.g. ADV(1..8))			
		Read ADV cell (ma) access signature table		
			Data items	Strong/weak flag (±)
				Control value Φ (ma)
				Ls (ma) = Left strength
				Lu (ma) = Left use
				Rs (ma) = Right strength
				Ru (ma) = Right use
				Rev(ma) = Reversal count
				Adv(ma) = advisor duty

count of the number of times any update process has reversed the cell's data value, and how many times the cell was called upon to give advice during the previous run.

The advisor process can be part of the end of run update schema or done completely on-the-fly during an active run. The data terms in Fig. 12.4 can be coded to produce left or right advice factors for consideration by the active cell before it returns its natural control value. Regardless of the advising scheme to be adopted, if the advice offered by the advisor actor is greater than the current persuasion of the active cell, its control decision for that situation is adjusted accordingly. If the advice appears weak or indecisive, then its contribution is ignored. The following section describes two different schemas and a possible third further overall improvement to advisor methodology in general.

12.2 Advising Schemas

The selection and ranking process is designed to aggressively discourage states from offering bad advice using positional and statistical rules. For example, a very strong near advisor may be able to assist a naïve or weak cell and override its original random value and consequently promote more rapid learning. However, if the natural cell is correct in its current data value and a *bullying* neighbour causes it to flip its data, the effect is reversed. With this in mind, there are several advising schemas each containing different sets of rules that can be applied. Three schemas are presented here; advising by voting, strength-based advising, and delayed advising.

12.2.1 Advising by Voting

A simple advising scheme lets each advisor cell determined according to the process described in Fig. 12.2 to vote based on the control value (u) of the active cell. Before a control decision is returned for any state, m, the system selects and examines a set of advisor states ($ADV\{1\ldots na\}$) and inspects the decision each would have returned had it been the natural state of choice. The number of votes allocated to the advisor cell is based on its relative position, and the advisor system tallies up the votes from each advisor. The natural state vote added into the tally and can be given preferred voting power by allocating it an appropriate level of substantial shareholder votes. Using a *winner takes all* philosophy, the votes for each control value (LEFT or RIGHT) are counted and the winning control value returned accordingly.

In the event of a tie, the value corresponding to the natural vote prevails. This is equivalent to a consensus building process. A variation on this method is to assign the natural state more than one vote in order to reduce capricious advising especially in the early part of a learning campaign, as will be illustrated later in Sect. 12.2.3.

It is straightforward to implant this feature into the standard algorithm, provided the edges of the state space are handled correctly. If a net negative voting summation result ensues, a LEFT decision would be returned, regardless of the cell's true value, and vice versa if positive. If the eventual merit decreases, the advisors that overturned the natural vote must be statistically punished.

Recalling that the matrix values are usually signed to represent LEFT/RIGHT motion, the weighted control decisions contribute their vote by algebraic summation using the following pseudocode and Eq. 12.5.

```
'Active cell enters its natural vote
        vote = $\Phi$[m] * Nv
' Advisors enter their (signed) votes
        FOR each advisor, j $\in$ ADV{1...na}
vote = vote + $\Phi$[j]*w(j)
        NEXT advisor
'Return plus or minus (RIGHT/LEFT) decision if vote not zero
                IF vote $\neq$ 0 THEN $u$ = SIGN[vote] ELSE $u$ = $\Phi$[m]        (12.5)
```

where,

Nv	active cell voting allocation
na	# of advisors proximate to cell m
ADV { }	the set of na advisor state numbers
w_j	vote allocation for advisor *j*
u	control decision (+1 or –1)

In the voting method above, advisor cells may be given more than one vote to reflect their proximity to the natural cell and/or their experience and strength as reflected in the weight value.

12.2.2 Advising Using Cell Strengths

A second, more computationally intensive, method performs a similar scan of all possible advisor states, but instead of returning a simple "vote", an aggregation of the decision strengths of the selected advisors was used to offer advice to the true cell. The following pseudocode and Eq. 12.6 describe the process more exactly for a natural cell (*m*). It is assumed that the strength and experience of each cell in the signature table are known as indicated in Fig. 12.3.

In this method, the signed advisor strengths, weighted by some positional factor are added to the active strength values and the result examined for its sign. As always, a positive result would cause a RIGHT motor direction to be asserted and vice versa. If the advisor sum was zero, the advice is ignored and the natural cell value returned.

```
'Retrieve signed strength of natural cell
  IF $\Phi[m]$ = "L" THEN
        strength = $LS_m * \Phi[m]$
    ELSE
        strength = $RS_m * \Phi[j]$
    ENDIF
'Process advisor strengths
  advice = 0
  FOR each advisor, $j \in ADV\{1...na\}$
  IF $\Phi[j]$ = "L" THEN
        advice = advice + $LS_j * w_j * \Phi[j]$
    ELSE
        advice = advice + $RS_j * w_j * \Phi[j]$
    ENDIF
  NEXT advisor
'Return u value based on sign of strength
      sum = strength + advice
```

$$\text{IF sum} \neq 0 \text{ THEN u} = \text{SIGN (sum) ELSE u} = \Phi[\text{m}] \tag{12.6}$$

Both schemas work well and Fig. 12.5 shows a comparison of the performance of the two methods as compared to the original BOXES algorithm.

The second scheme is called statistical advising because it relies upon the statistical strength (knowledge) of each of the advisor cells included in the determination of possible changes to the value of the natural control variable. This is equivalent to a collaborating team approach to learning inside the algorithm.

12.2.3 Delayed Advising

It is evident from Fig. 12.5 that the simple voting method exhibits a much greater learning rate than that exhibited by the statistical method. On reflection, the data in the signature table is initially random so the values of left and right strength are somewhat arbitrary. It is proposed that the advising process be delayed until the original algorithm has had time to mature and better results are to be expected.

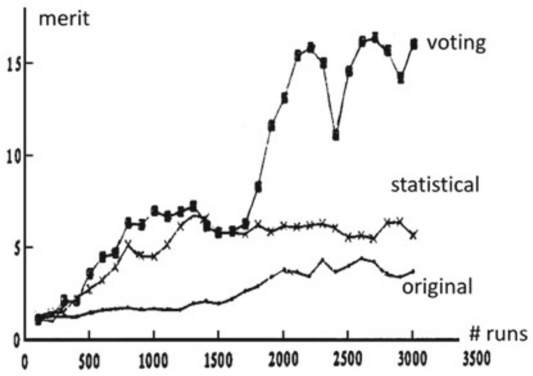


Fig. 12.5 Comparison of voting and statistical advisors

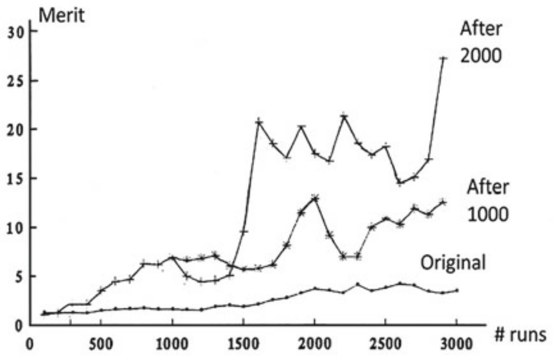


Fig. 12.6 Delayed advisor results

Figure 12.6 shows results from a typical learning run for the modified BOXES algorithm [6] applied to a simulated trolley and pole system simulation and appears to support the notion that delaying the advising process encourages better learning later.

At the end of a run, the advisor states must be made accountable for their contribution to decisions made inside the run. Because matrix values and cell statistics are fixed during a run, it is only possible to use current run values, advisor overrides and the merit of the system off-line to achieve this result.

12.3 Advisor Accountability

At the end of every learning run values for system merit are recomputed to reflect an overall performance or skill level for the system. To effect learning, the BOXES standard post-processing algorithm updates the statistical database for each cell by merging the data accumulated during the previous run with statistics stored from

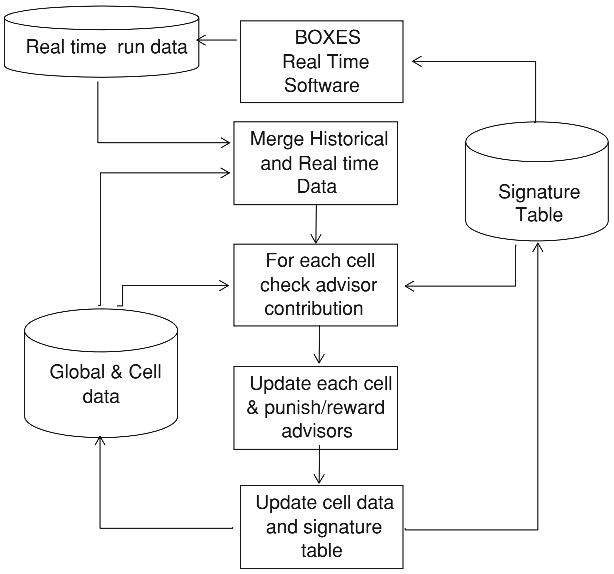

Real time run data
BOXES Real Time Software
Merge Historical and Real time Data
Signature Table
For each cell check advisor contribution
Global & Cell data
Update each cell & punish/reward advisors
Update cell data and signature table

Fig. 12.7 Accountable advisor logic

all previous runs. Once the statistical database has been updated to include the current run and any reward or penalty scores related to its advising duties, the algorithm computes new values for its left/right statistical strengths and assigns new values as necessary to the signature table data. Advisor cells can greatly influence the controllability of the system as explained above. It would appear reasonable to make the advisor cells accountable for their actions as part of the update process. Figure 12.7 indicates the logic necessary to achieve this.

A cell might provide advice to many other cells during the course of a controlled run causing the accountability process to be highly recursive. It is the change in performance that is important as a campaign of runs progresses. Equation 12.7 is a simple method of estimating this and Q reflects the system's performance gain at the end of a run and is a good starting place in evaluating advisors.

$$Q = (\text{new\_merit} - \text{old\_merit}) / \text{new\_merit} \tag{12.7}$$

This is used in conjunction with every state's advisors and their ranking, in modifying historical statistics as follows:

For each state, *m*

```
'Produce ADV {1...na} list of advisors
  For each advisor ADV(ai)
IF u(ai) = "LEFT" THEN
```

$$L\_\mathrm{timdat}_{\mathrm{ai}} = L\_\mathrm{timdat}_{\mathrm{ai}} * \mathrm{F} * (1 + \mathrm{Q}) * \mathrm{w(ai)} \tag{12.8}$$

```
IF u(ai) = "RIGHT" THEN
```

$$R\_\mathrm{timdat}_{\mathrm{ai}} = R\_\mathrm{timdat}_{\mathrm{ai}} * F * (1 + \mathrm{Q}) * \mathrm{w(ai)} \tag{12.9}$$

```
  Next advisor
Next state.
```

where,

Q	Merit factor
F	An overall scale factor
$w(ai)$	Advisor weight for cell "m"
$L\_timdat(ai)$	Lifetime statistical term when decision is LEFT
$R\_timdat(ai)$	Lifetime statistical term when decision is RIGHT

If this is applied, the statistics are altered and now include a "penalty and reward" measure of accountability as an increase in the value of Q causes the lifetime data to be increased because of good advice given to other states and vice versa. If an advisor cell was not entered during any given run, its experiential data is left unaltered.

During the post-processing phase, cells chosen to be advisors are statistically rewarded or penalized based on their contribution to the change in system merit. The change in lifetime data (*timdat* in Eqs. 12.8 and 12.9) for their current signature table value alters their strength values during the final update phase. In this manner the advisors are made statistically accountable for their influence on the native cell value. This was seen as necessary in order to respect the strength of mature cells and to incidentally add a secondary heuristic to the learning process as the advisor learns from the advisee which is equivalent to what is known as 360$^{\mathrm{o}}$ evaluation. Adding advisor logic to BOXES software adds quite considerable complexity.

12.4 Conclusion

This chapter indicates that the BOXES black-box signature table control paradigm can be extended to reduce the dependency on the decision contained in any unique state using near neighbor advisors. After each control sequence, the advisors are statistically rewarded or punished based on advice given to an active cell.

If the notion of peer advising is to be considered, each state must maintain advising data in addition to its normal counter and timer data. Care must be taken to select, screen, and validate advisor states before including them in the process and also ensure that they can be held accountable statistically for any advice rendered.

On a cautionary note, advising can cause a group of strong decision cells to permeate the control matrix, and render the individual (natural) state impotent. In order for this majority rule effect to be nullified, the researcher must be prepared for the appearance of rather disturbing groups of untrainable, ambivalent states, anywhere in the signature table.

The selection and ranking process is designed to aggressively seek to discourage states from offering cavalier advice for which they are not held accountable. The simpler voting scheme seems to offer a good method to improve the rate of learning of the system while the statistical schema seems to promote slower learning. This is not totally unexpected seeing that the initial control matrix is randomly generated. The lack of cogent data will render any statistical advising useless so delaying the advisor tactic for several thousand excursions of the system until the system control matrix has some maturity has merit when some startling performance improvements are apparent.

The implementation of advisor logic into the BOXES algorithm is an innovation that has proved to increase its rate of learning extremely well.

References

1. Russell, D.W. and Rees, S.J. 1975. System Control—a Case Study of a Statistical Learning Automaton. *Progress in Cybernetics Systems Research* 2:114–120, New York: Hemisphere Publishing Co
2. Rozonoer, L. (1959). Pontriagin's Maximum Principle in its Application to the Theory of Optimal Systems. *Automation Remote Control* Vol.20
3. Castillo, E. & Alvarez, E. (1991) *Expert Systems: Uncertainty and Learning*. Elsevier Applied Science. New York p.177
4. Russell, D.W. (1993) A Critical Assessment of the BOXES Paradigm. *J. Applied Artificial Intelligence* 7(4): 383–395
5. Russell, D.W., Super-boxes - an Accountable Advisor System (1993) *Proc. AIENG'93 8th Intl. Conf. on Applications of AI in Engineering*, Toulouse, France. Elsevier Press, 477–489
6. Russell, DW (1995) Advisor Enhanced Real Time System Control. *Control Engineering Practice*. Vol. 7: No.3 1995 Pergamon Press, 977–983

Chapter 13
Evolutionary Studies Research

13.1 Introduction

The BOXES methodology learns to adjust and implant increasingly profound control decisions into a signature table so that even ill-defined dynamic systems can access it and retrieve decisions for use in system control. The method surprisingly needs little a priori knowledge of the system under control beyond limiting values of the state variables and how each is to be divided into integer regions. Unlike most control schemes, the mathematics of the system does not feature in the control algorithm.

In order for this category of controllers to work, the span of each state variable is divided into equal or non-equal zones or regions of operation so that each state variable can be easily transformed into a *state integer* at any time. The BOXES method allows flexibility in the selection of how many and what size state variable zone boundaries may be. The division of the four state variables in the trolley and pole system into 225 [$5 \times 3 \times 5 \times 3$] zones has its roots in Michie's work [1] but other combinations are acceptable and have been used in other applications as shown in previous sections of the book. The debate is to whether many small regions are better than a few large ones. The system designer might choose a large region at either end of each state variables range and then subdivide across the middle region. Woodcock et al. [2] utilized a scheme to fuzzify the boundaries between zones in an attempt to reduce the significance of the boundary values thus blurring the state in which the system was operating. This was shown to produce some very long controlled runs, but did introduce hitherto unprecedented erratic dynamic behaviors while learning.

This implies that the optimal solution must lie in the zones that straddle the origin of each variable which is somewhat intuitive for the trolley and pole as it swings back and forth inside its limit cycle. In other systems this is probably not true.

Table 13.1 Effect of the number of boundaries on merit

Situation [boundaries] = zones	Merit after 200 runs	Merit after 400 runs	Merit after 500 runs
Standard $[5 \times 3 \times 5 \times 3] = 225$	60	350	480
Finer in dΘ/dt only $[5 \times 3 \times 5 \times 12] = 900$	10	15	16
Finer in Θ $[3 \times 5 \times 8 \times 4] = 480$	5	7	9
Finer in Θ and dΘ/dt $[3 \times 2 \times 16 \times 10] = 960$	12	24	26

As will be shown in Sect. 13.2 large variations in values produce inconsistent and unremarkable results. Section 13.3 introduces a method that nudges boundary values from their initial position based on how the system is improving. Section 13.4 describes an algorithm to accomplish the task.

13.2 State Boundary Experiments

During the Liverpool experiment described in detail in Chap. 6, some research was devoted to the effects of altering the number and location of the boundary [3] positions.

13.2.1 Variation in the Number and Size of State Zones

Simulated runs were made varying the number of boundaries for the state variables. Table 13.1 shows the effect on merit over the same number of runs of varying the number of boundaries. The numbers in square brackets are coded as: [#x zones, #dx/dt zones, #Θ zones, and #dΘ/dt zones].

These data in Table 13.1 are typical experimental values and included as indicators only. Interested programmers are encouraged to perform their own tests and confirm that the learning dynamics do indeed vary as the boundaries are changed.

13.2.2 Preliminary Conclusions

These tests indicate that the number of boundaries seems very important to learning. Of course, the number of states is given by the product of the number of state variable zones. With this in mind, one reason for the poor merit figures with more zones might be that a lower percentage of states are entered during a sample 500 run campaign. The large difference between the cases where the number of

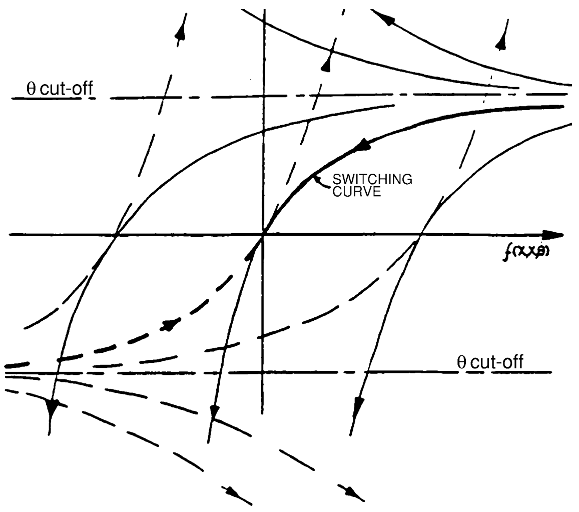


Fig. 13.1 Switching curve for a trolley and pole

boundaries was varied and the original is noteworthy. It may be that control of the trolley and pole coincidentally prefers the five even sub-divisions of (x) and (Θ) and a small central zone for (dx/dt) and (dΘ/dt) in much the same way that a fuzzy system is formed.

13.3 An Evolutionary Paradigm

Because the method by which the state variables are segmented into zones is arbitrary, based on Sect. 13.1, any notion of self-organization will not allow alterations to the number of zones in the system. This enables the proposed system to alter the physical dimensions of the hypercubes that make up the cell structure, but will not change the number of them. On reflection, this morphological process attacks the uncertainty associated with the location of the cell boundaries, which is a potential flaw in the BOXES paradigm. Another benefit, and the focus of future research, may be that as the proposed method alters the cell size the creation of small untrainable zones may indicate the location of the switching surface. Figure 13.1 illustrates a typical switching (control) curve [4] for the trolley and pole.

13.3.1 Types of Zones

The mathematical solution of the trolley and pole equations yields the motion pattern of an inverted pendulum. This pendular motion is expressed by the trolley oscillating up and down the track in harmony with the pole that swings freely about a central pivot on the trolley. This implies that there will be a set of BOXES

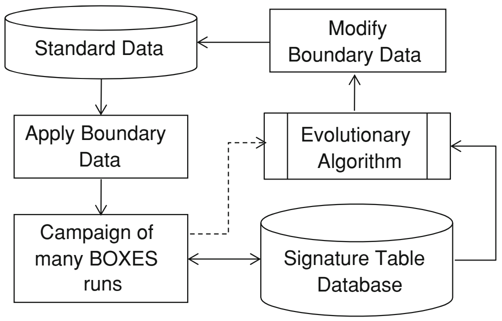


Fig. 13.2 Amendment to the BOXES software architecture

states that are entered and exited often during control runs. It is expected that over time these cells should become stronger and more certain in their decision values.

Other cells will be less certain and grow weaker in their decision strengths. And there is a set of other zones that will probably be entered rarely as they lie outside of the normal control space. It should be noted that in the Liverpool system, the trolley did exhibit precession up and down the track forcing exploration of otherwise non-contributory states. This led to some BOXES systems dictating that extra rewards be given to runs in which the trolley crosses the mid-point of the track or the pole swings across the vertical. The idea of this was to keep the system focused on the central region of the track.

13.3.2 An Evolutionary Method

As the system matures, it is proposed that an algorithm is possible that can mine the signature table statistics that have accrued for all cells. Figure 13.2 shows how the original BOXES architecture would need to be amended to include this feature. The standard data repository includes the boundary values to be applied in the BOXES algorithm. The evolutionary algorithm that can modify these values is triggered only after some large number of runs.

The evolutionary algorithm classifies each BOXES cell in regard to its strength and certainty and computes a possible positive or negative *nudge* value that is applicable to a near boundary. If valid, it is allowed to nudge the boundary by that amount for future runs. After many excursions of the system, some idea as to where better locations of the boundaries should become apparent. Figure 13.3 shows the effect that the algorithm is designed to have over time.

In this example, the boundary between the BOXES cell variable designated (i) is nudged toward its neighbor ($i + 1$) by a small computed amount (ΔB). The movement shown to the right in Fig. 13.3 was produced by the hypothetical case that the original cell (i) is very mature and stronger than its neighbor ($i + 1$) regardless of what its decision value might be. It is clear that this method could conceivably cause very strong cells to completely dominate the assignment of boundaries for any state variable, so some limitations must be imposed as follows.

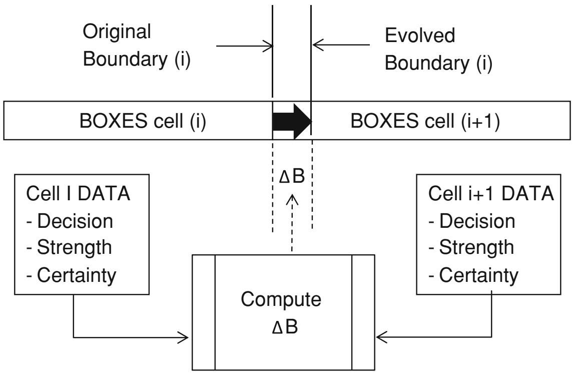


Fig. 13.3 Boundary nudging using statistical cell data

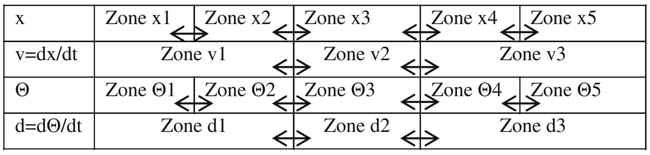


Fig. 13.4 Moveable zones for the [5 × 3 × 5 × 3] trolley and pole

13.3.2.1 Zone Discipline

For the algorithm to be effective, a limit is set on the maximum value of any nudge (ΔB) value. It is also prudent to set up a minimum zone size so that the production of algebraically negative or miniscule zones is avoided. The end positions (zero and one) cannot be moved, so the number of moveable boundaries is always the same. If the zones were to be coalesced or subdivided, a complete reconfiguration and restart of the algorithm would be necessary as the signature table and its data would be violated.

For example, Fig. 13.4 shows in the [5 × 3 × 5 × 3] configuration discussed above that there are only 12 moveable boundaries that can be nudged either way. Over time erroneous boundary movements are nullified by subsequent actions of the algorithm in much the same way as humans can change their minds with experience. Quantifying the strength and certainty of a cell value which is necessary for this evolutionary nudging process is described in the next section.

13.3.3 Interpreting Signature Table Statistics

As described in Part II of this book, statistics are maintained for every box or cell in the configuration. This data was originally used purely to update the signature table decision data values (LEFT/RIGHT) and was based on global data after temporary values from the immediately past run were incorporated. For the trolley and pole system there are LEFT and RIGHT use and strength data, but in other systems there are other values, such as ON or OFF. In some simulations more than two signature table token values were included that represented three motor state signals; LEFT, COAST, or RIGHT. Because the optimal system is of the bang-bang variety it was not surprising that the results were not favorable.

13.3.3.1 An Original Use of Data

Chapter 5 explains how, during a control run, a count of how many times a cell (or *box*) is entered and a running sum of the times at which the cell was entered is kept as non-persistent data. At the end of the control run, a value of system merit which is a measure of the level of achievement of the controller is evaluated. Once this is known, the strength of each decision can be computed for each of the system integers using Eqs. 13.1–13.3 which are repeated for clarity from Eqs. 5.14–5.16.

$$\text{dla} = C_0 + C_1 {}^* \text{merit} \tag{13.1}$$

$$\text{left\_strength}_i = \frac{(\text{left\_life}_i + K^*\text{dla})}{(\text{left\_count}_i + K)} \tag{13.2}$$

$$\text{right\_strength}_i = \frac{(\text{right\_life}_i + K^*\text{dla})}{(\text{right\_count}_i + K)} \tag{13.3}$$

Based on these values, the control values in the signature table for each index (box) value are reset to *LEFT* or *RIGHT* in the update phase of operation. What is important to note is that not only does this data give decision strength, but may also indicate the experience level of any given state, because a sparsely used state will have low *count* values that may produce high strength metrics. Over time, this effect evens out.

13.3.3.2 Measuring Decision Certainty

What is proposed for the evolutionary method is that existing signature table data can be manipulated offline to give some measure of decision certainty based on the historical signature table data only without any reference to the current merit of the system as is needed in Eq. 13.1. A proposed estimate of the certainty of any decision can be made for all cells (i) that have been entered at least once is given by Eqs. 13.4–13.6.

$$\text{LS}_i = \text{Llife}(i)/\text{Luse}(i) \tag{13.4}$$

$$\text{RS}_i = \text{Rlife}(i)/\text{Ruse}(i) \tag{13.5}$$

And, provided the denominator is not zero,

$$\mathrm{CY}_i = \mathrm{ABS}\ (\mathrm{LS}_i - \mathrm{RS}_i)/\,(\mathrm{LS}_i + \mathrm{RS}_i) \tag{13.6}$$

where

LS_i	Left decision average lifetime per entry of cell (i)
RS_i	Right decision average lifetime per entry of cell (i)
CYi	Certainty of current decision u {= $\Phi(i)$}

This certainty is now used as a weighting factor in estimating the true strength of any box or cell. In that event, Eqs. 5.17 and 5.18 (in Chap. 5) would be combined to give Eqs. 13.7–13.9 and exhibit faster learning in the original algorithm.

$$\text{IF}\,\mathrm{CY}_i > 0.5(50\%)\ \text{THEN NO ACTION}\ \{\text{leave (u) unaltered}\} \tag{13.7}$$

$$\text{IF}\,\mathrm{CY}_i < 0.5(50\%)\ \text{THEN REVERSE}\ \{(\text{u})\} \tag{13.8}$$

$$\text{IF}\,\mathrm{CY}_i = 0.5(50\%)\ \text{THEN NO ACTION}\ \{\text{or RANDOMISE}\} \tag{13.9}$$

While this looks rather simplistic in the original algorithm, it has a profound effect on advisor modification as described in Chap. 12 is applied because only advisors that have high certainty factors would be considered as potential advisors. However, a cell can posses a high certainty figure and still be incorrect in the long term.

13.3.3.3 Evolutionary Boundary Neighbors

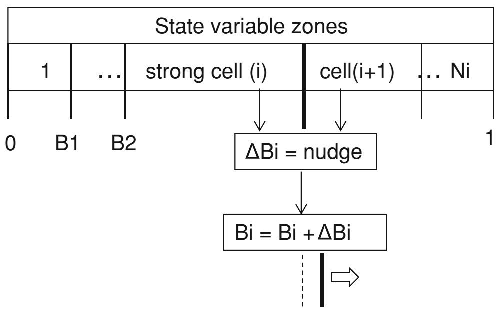
When considering nudging a boundary, the problem can be simplified by considering cells on a state variable by state variable basis. Figure 13.3 shows how the process for two cells (i) and (i + 1) can influence the boundary value. Figure 13.4 shows how each state variable only has a few opportunities for movement and that any boundary movement needs to reference and will influence other neighboring states. For consistency, Fig. 13.5 illustrates how any movement should only be applied to the boundary to the right of any cell.

In a multi-variable BOXES application, moving the boundary (B_i) will influence a group of states which are identified by Fig. 13.6 for the [5 × 3 × 5 × 3} configuration for the state integer i.

13.3.3.4 Aggregate Nudge Factors

The following is an outline of the pseudo code that would enable all of the moveable boundaries to be moved after some long sequence of runs.

Fig. 13.5 Cell boundary relationship



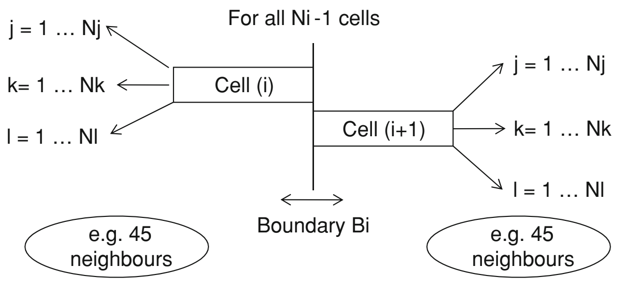


Fig. 13.6 Evolutionary advisor states for one state variable

```
‘Scan moveable zone boundaries for the state integer “i”
  FOR i = 1 to Ni-1
       ΔB = 0
‘Select all neighbors in turn for cell i and i + 1 using function MVALUE
       FOR j = 1 TO Nj
            FOR k = 1 TO Nk
                 FOR i = 1 TO Nl
                      MVALUE (m, i, j, k, l)
                      MVALUE (n, i + 1, j, k, l)
‘Check if cell and neighbor are valid state variables
                      IF m AND n 0 THEN
‘Compute nudge value using function GETΔB
                           GETΔB (m, n, nudge)
                           ΔB = ΔB + nudge
                           ENDIF
                 NEXT l
            NEXT k
       NEXT j
‘When all neighbors have been considered nudge this boundary by ΔB
       BI(i) = BI(i) + ΔB
  NEXT i
```

The sub-programs MVALUE and GETΔB are defined as:

MVALUE:	Computes the system integer from four state integers
GETΔB:	Computes the nudge value between cells “m” and “n”

The same process would need to be repeated for the other state integers and a new campaign of BOXES control runs initiated using the modified values for state boundaries.

13.3.4 An Evolutionary Algorithm

The function titled “GETΔB” would include an algorithm such as:

```
FUNCTION GETΔB {cell1, cell2, nudge}
nudge = 0
‘Use Eqs. (13.4–13.6) to get strength and certainty of cell1
    LSm = Llife(cell1)/Luse(cell1)
    RSm = Rlife(cell1)/Ruse(cell1)
    CYm = ABS (LSm − RSm)/(LSm + RSm)
‘Repeat for neighbor – cell2
    LSn = Llife(cell2)/Luse(cell2)
    RSn = Rlife(cell2)/Ruse(cell2)
    CYn = ABS (LSn − RSn)/(LSn + RSn)
‘Compute a nudge factor using these two contributors, using Eqs. 13.10–13.12
```

$$\text{IF}\,\Theta(cell1) = \text{''L''}\ \text{THEN}\,SM = LSm\,\text{ELSE}\,SM = RSm \tag{13.10}$$

$$\text{IF}\,\Theta(cell2) = \text{''L''}\ \text{THEN}\,SN = LSn\,\text{ELSE}\,SN = RSn \tag{13.11}$$

$$nudge = BF * (SM * CYm\ --SB * CYn)\,/(SM + SN) \tag{13.12}$$

```
RETURN {nudge value}
```

where, BF is a factor limiting the size of any boundary movement forced by any two neighbors. The reasoning behind Eq. 13.12 is to modify the strength of any cell based on its experience and certainty value. In this manner, cells that are incidentally or maybe even randomly given strong values with little game experience will be down rated. If the strength–certainty ratio is identical for both cells, there will be zero boundary nudging. The algorithm will allow nudge values between plus and minus BF, so positive and minus net changes will balance out.

Table 13.2 Three numerical boundary change examples

Advisor (n) state data Φ(n) = LEFT	Natural (m) state data Φ(m) = RIGHT	Equation (13.12) gives ΔB	Boundary effect
Sn = LSn = 4 CYn = 3 STRONG "L"	SM = RSm = 4, CYn = 2	+0.5*BF	Advisor encroaches on natural boundary by 0.5
SA = LSn = 4 CYn = 3	SM = RSm = 6, CYm = 4 STRONG "R"	−1.2*BF	Natural encroaches on advisor boundary by 1.2 units
Sn = LSn = 4 CYn = 3 STRONG "L"	SN = LSi = 3 CYm = 4 STRONG "R"	0.0	No boundary shift—as neither state is definitively stronger

13.3.5 Example Results

A small program was written to exercise the algorithms contained in Eq. 13.4 through Eqs. 13.6, and 13.10 through Eq. 13.12 using random data to illustrate the facility of the proposed system [5] and that reasonable results can be obtained. Table 13.2 contains three numerical examples for a trolley and pole system and how each would influence a boundary value.

The control decision values (LEFT/RIGHT) only feature in the selection of which data sets are to be used in the boundary calculations. In the normal update procedure these determine the next data value to be set.

13.4 Conclusion

This chapter is the subject of ongoing research and interested readers are encouraged to produce such a system. As the prior sections have shown, it would appear that the location of the boundaries within each state variable is important, and yet must not be over-emphasized to the extent that they become identifiable as a priori knowledge. Some studies have used a genetic algorithm in an attempt to locate optimal boundary value positions. It is apparent that proximate system integer regions can positively influence the learning curve of the system over time. The chapter has proposed that a solution may lie within the statistical database after many runs. To take advantage of this, the BOXES controller, which is of the black-box signature table variety, may be extended to include boundary movement in order to reduce the dependency on how any individual state integer is produced. It would appear that while both methods improve on the original BOXES method, the voting modification accelerates learning progress beyond that of the cell-by-cell statistical strength method. Because the delineation between state integers is completely dependent on the location of the interstitial boundaries inside each of the normalized state variables, it is proposed that much more useful information exists within the signature table database than was first thought.

This proposed adaptation of the BOXES method is intriguing in that the principles presented may contain a metaphor of how the human adaptive cognitive process handles and differentiates between learning and certainty. Most models of the brain have difficulty with the explanation of how humans process “knowing for sure” and “believing to be so” and how decisions based on both vary over time and after practice. An accumulation of skill or dexterity in some area reduces the cognitive effort and time to produce a preliminary action plan for a similar related task, although the new task may be much more complex and well include a search space that is larger than before. A novice may not have this advantage and without the ability to apply boundary knowledge, find the task impossible. It is interesting to compare the popular metaphor of cellular synaptic connectivity [6] to an ideology of cooperating neural regions.

References

1. Michie, D., and R. A. Chambers. (1968) BOXES - an experiment in adaptive control. In *Machine intelligence* II, 2, 137–152. London: Oliver and Boyd.
2. Woodcock, N., Halam, J.J. and Picton, P.D. (1991) Fuzzy BOXES as an Alternative to Neural Networks for Difficult Control Problems. Applications of Artificial intelligence VI (Proceedings AIENG’91 Oxford) 903–920. London: Elsevier.
3. Rees SJ Thesis (1978) – Liverpool Polytechnic. Chapter 4
4. Russell, D.W. and Rees, S.J. (1975) System Control—a Case Study of a Statistical Learning Automaton. Progress in Cybernetics Systems Research 2:114–120, New York: Hemisphere Publishing Co.
5. Russell, D.W. (2004) On the Control of Dynamically Unstable Systems Using a Self Organizing Black Box Controller. *Proc 7th Biennial ASMA Conference on Engineering Systems Design and Analysis*- ESDA 2004, Manchester, UK, July 19–22. CD
6. Chklovskii, DB (2004) Synaptic connectivity and neuronal morphology: two sides of the same coin. (2004) *Neuron.* 43(5):609–17.

Chapter 14
Conclusions

14.1 Some Philosophical Commentary

Intelligent systems, such as those at work in the human mind, exhibit varying degrees of awareness, skill, savvy, adaptability, gifting, and genius. Doyle cleverly distinguishes the brain from the mind. In an essay [1] devoted to the concept of mind and the positioning of artificial intelligence within cognitive science, he asserts that adaptiveness is an important component of intelligence. The ability to change beliefs can be based on routine updates in information.

It is widely accepted that learning is contingent upon, among other factors, the environment in which it is taking place and how that environment is altered by outside influences over which the learning agent has little or no control. The environment includes motivational forces, demands, critical assessments, and corrective actions designed to assist in subsequent courses of action. Baron [2] suggests that if motivational and evaluative data is provided to human agents about their performance, then it is reasonable to expect that some attempt to improve will be made. The very nature of the terms *motivational* and *improve* implies that some acceptable measure of performance has been made and that a mechanism for correction or recovery is available and has been presented. Of course, the agent must accept the accuracy of the metric and be willing to investigate where changes should be made. It is a human trait to exhibit bounded rationality [3] in which facts *and* emotions influence decisions. Because of this, in an attempt to meet criteria for adequacy, rather than optimality humans are often willing to settle for an acceptable solution to a problem. This is known as *satisficing*.

If proper learning is to occur, the teacher or critic must recognize features or trends in how the student is performing and then be able to convey correct and timely feedback in such a manner as to convince the pupil of the need for change. If the student fails to grasp the import of the advice, is distracted, or simply chooses to ignore it, the learning process is thwarted and may even become corrupt and inept. When information is missing, humans resort to inductive logic in which past experiences, gut feelings, recognizable patterns, or familiar models take

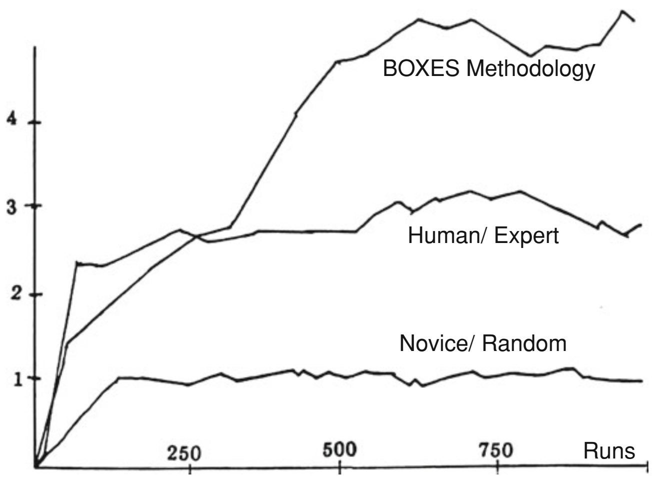


Fig. 14.1 Performance comparison of human and automaton operators

precedence over purely fact-based reasoning. Good chess players exhibit inductive behavior constantly as shown by their adoption of well-known [4] defences of other players during the opening phase of a game.

An important facet of learning is the ability to remember and selectively forget facts and events and be able to synthesize and recall timely information. Rose [5] suggests that the human ability to forget is essential for survival and possibly even less destructive than our capacity to remember.

If the BOXES method is purely a system of pragmatic guesswork that lands upon solutions by happenstance, then the signature table data, and indeed all payoff matrix systems, are meaningless.

Many of the features that appear in human learning can be programed into methodologies such as the BOXES algorithm and are essential in the formulation of systems that (apparently) learn to perform tasks of varying difficulty, but with one exception. Automata powered by software such as the BOXES method do not exhibit bounded rationality, rather they accept performance metrics and supplied advice, and make adjustments without question. In any specific task, an automaton may far outpace the efforts of human novices or experts. The reason for this superiority might lie in their ability to repeat assembly tasks endlessly without interruption, to perform tasks in environments that are too hazardous for human entry, or to react at speeds that far outpace human reflexes.

Figure 14.1 illustrates this using real data taken during the Liverpool trolley and pole experiments.

Clearly the environment should be conducive to learning, but care must be taken to avoid the assignment of any teleological implications in generating intent or advice from an automaton. The computer system is purely deductive and numerically based without any notion or measure of reality outside of its programed interfaces. Conversely, an inexperienced human tutor may give partially correct, but totally irrelevant, advice to a student and thus assert a negative influence that may reach far beyond the immediate.

Mapping the student-teacher analogy into the realm of adaptive control of a dynamic real-world system, similar problems and associated risks are compounded by the automaton's lack of inherent common sense, life skills, and intuition which are among the tools that human operators can always fall back on. However, automated responses may be mathematically and logically correct but still be dangerous or impossible to implement in the real world.

But an automaton does have several distinct advantages over a human agent. It performs given tasks with undivided attention, absolute concentration, and loyalty, and operates at speeds far in excess of those of human reflexes. Once configured correctly, automata have very modest needs beyond a clean power source and occasional lubrication. As shown in Fig. 14.1 human operators did not accomplish long pole balancing times with or without hours of training. The BOXES system rapidly outperformed the human without fatigue.

Chapter 1 served to introduce the concept of machine intelligence using familiar game-playing examples, and how scientists and engineers often approach similar problems from two distinct standpoints. Modern digital computers are unimaginably fast, data retentive, and relentlessly focused on performing a stream of instructions. Are they merely doing what they have been told to do or are they capable of learning to perform meaningful tasks?

14.2 Bloom's Taxonomy

In 1956, Benjamin Bloom [6] headed a group of educational psychologists who developed a classification of levels of intellectual behavior that are important in understanding the process of learning. From lowest to highest the Bloom levels are:

1. Learning:	The recall of memorized information.
2. Comprehension:	The interpretation of meaning and how it relates to the future.
3. Application:	The ability to meaningfully apply what has been learned to new situations or problems.
4. Analysis:	The comprehension of the organizational principles that constitute parts.
5. Synthesis:	The ability to put together disparate parts to form a coherent whole.
6. Evaluation:	The introduction of value judgments.

The following Sects. 14.3–14.6 are chapter-by-chapter summaries of the contents of the book to this point; each is designed to suggest if any of these learning criteria are achieved by the BOXES method and to what extent those rules apply.

14.3 Part I: Learning and Artificial Intelligence

Chapters 2, 3 and 4 of the book were an introduction to the type of computational first (memory) and second (futures) levels of learning that computational systems can be designed to play board games, and introduces how the BOXES method achieves success while playing the real-time control using the *game* paradigm.

14.3.1 Chapter 2

In the introduction quite some time was spent describing how the rules of chess which is a fairly sophisticated but still deterministic game can be programed into a digital computer. This chapter described some other somewhat simpler games that can also be programed to be played increasingly well by the computer. While all computer games are not necessarily alike they vary mainly in the concepts of game domain, winning and losing, and the mechanics of game play. Because game strategies differ, it is possible for a computer program to learn new games based purely on statistics and while certainly not crafty, they can do justice to the sport.

It is not a coincidence that the fundamental premise of the BOXES method is that *dynamic control can be played as a game,* and that *winning is achieved by staving off defeat or failure.* The methodology inherent in the BOXES algorithm falls into five major components; defining the game board, identifying the location and type of game pieces, codifying legal moves for those pieces, end-game detection, and finally, the creation and enforcement of strategies for improvement. These are the same components necessary for most other games.

14.3.2 Chapter 3

This chapter described the first version of the BOXES paradigm that Michie introduced under the acronym MENACE which stands for *Machine Educable Noughts And Crosses Engine.* Using a simple array of matchboxes, it was possible to play noughts and crosses using no other strategy than to add or remove colored beads in each matchbox based on the outcome of each game.

In real-time control, instead of a *player* reading the current board scenario and selecting a move, the BOXES algorithm must be provided with the current state of the system which it codifies into a system integer which is used as the pointer that reads off a control value from a signature table and the real-time system progresses. The mechanically unstable trolley and pole system was selected as an ideal candidate for system proving.

The BOXES algorithm operates in two phases: reactive in real-time closed loop control; and reflective by learning after a run is terminated for any reason. In reactive mode, decisions are simply read from the signature table. In reflective mode, the signature table is modified based on data collected during a run that can be linked to the perceived effects of each cell's control value on the overall system performance. For the trolley and pole, this process requires a two-dimensional payoff matrix that is populated with statistical data related to past play in the control game. This data might be an encapsulation of the longevity of a run, the number of times the system entered each state or the severity of induced transient oscillations, and so on.

After all cells have been processed, the decision strength of each cell is recomputed and compared. The prevailing decision is then entered as the signature table value for use during the next run whenever that state is entered. Bloom's level 4 (analysis) is at work inside the algorithm in that real data are aggregated for each cell from an overall performance expectation.

14.3.3 Chapter 4

The BOXES algorithm can be configured to play dynamic control as a game. By adopting the notion of *dynamic control as a game to be won*, the same essential components of any BOXES implementation remain. This chapter focused on the real-world aspects of configuring a BOXES system and gave a more detailed coverage of the mathematical and statistical procedures necessary for learning to occur.

These included the collection of real-time data as the system moves from state to state repeatedly. This chapter showed that it was almost always necessary to count the number of entries into any state during a run as it reflects the experience level of that state which is related to how dependable that state is in its future activities.. The actual real data could be a summation of how long the system spent in that state, or some other game-related value recorded during the run. In the update phase it was necessary to age all statistical data as a means of reducing any unintentional bias in the random initialization process. An estimate of the strength of each decision was based on how the system just performed on the whole, and some computed quantity cell by cell that can strengthen or weaken each token's decision.

14.4 Part II: The Trolley and Pole

To illustrate the efficacy of the BOXES method, a trolley and pole system is regularly used because its mathematical model is well known, it is simple to build, and because it is controlled by a single LEFT/RIGHT motor signal. Chapters 5, 6 and 7 of the book focus on various more practical aspects of this particular application and address the differences and problems between simulated systems and the real world.

14.4.1 Chapter 5

Many systems are difficult to control by conventional means because of their complexity. Some systems perform very well most of the time, but under certain conditions exhibit unstable or even chaotic transient behavior. Such events occur in bioreactors, electroplating, and many other application domains. In these cases a model may not exist that can be trusted to accurately replicate the dynamics of the real-world system. To demonstrate the methodology, this chapter described how a simulated model of a two-dimensional pole and cart could learn to balance the freely swinging pole by moving the cart in rhythmic oscillatory motion. This system should not be confused with pole balancing in which the pole is swung up from a rest position and held in some sort of equilibrium by vibrating the cart under the pole.

As part of the simulations, the notion of near neighbor cells was introduced as possibly containing helpful advice to other cells over which they may have influence. Using simulation allowed many runs to be executed and some idea as to the learning potential of the BOXES method obtained. Proof of concept is suggested using results from several simulations.

14.4.2 Chapter 6

While the use of simulation in the development of algorithms is very useful and adopted frequently throughout this text, the real proof of any control paradigm is measured by its ability to handle real physical systems. This chapter focused specifically on the application of the BOXES methodology to real trolley and pole rigs constructed in Liverpool. The details are enough to the point of encouraging readers to construct their own versions of the rig. The control task is the same as that covered in Chap. 5 which is to investigate if a pole and cart system can learn to balance the freely swinging pole by moving the cart back and forth. This chapter described how an actual trolley and pole system was built and controlled by either a human operator via a manual override switch or by the BOXES algorithm which was implemented via custom-built hybrid computer firmware.

Overall, the merit values obtained are much lower in the physical studies than those obtained from similar simulation studies, but they did indicate that the automaton could learn to outperform humans very quickly. Several project persons operated the manual control switch regularly over a period of several years and although their performance did exceed that of a walk-in novice, the automaton produced better results after quite short training intervals. The computer system was obviously not distracted and exhibited Bloom's analysis (level 5) synthesis and evaluation.

14.4.3 *Chapter 7*

Chapters 5 and 6 illustrated that the BOXES methodology worked very well in prolonging the duration of motion of a freely hinged pole that is attached to a constantly moving cart. Chapter 5 considered simulations of the system that ignored the effects of friction in the hinge, while Chap. 6 illustrated the attachment of a real pole and cart rig to the algorithm using custom firmware and real-time data collection.

This chapter focused on how it was essential to ensure that training sequences preclude any preferential treatment of any region in the control space that might have been inadvertently injected during the random initialization process. To accomplish this, auto start systems were deemed necessary to control how the system should be parked after a failure, and subsequently restarted. During a restart sequence, the motor direction on the trolley may not necessarily agree with the corresponding value in the system signature table. In fact, control is bumplessly handed over to the BOXES automaton when a valid signature table value agreed with the current state of the system after a valid sequence of motion has been instigated.

It was deemed imperative that the system be not overtrained in any one trajectory, such as is the case if the pole were to be always held vertically with the trolley positioned in the center of the track before release.

In simulation studies it is relatively simple to ensure that initial values of the state variables are truly random from run to run and distributed across the vector space. In physical system, the autostart firmware was designed to promote truly random starts of the system and also allow a manual override of the control logic using LEFT/RIGHT push button commands.

14.5 Part III: Other BOXES Applications

Chapters 8, 9 and 10 of the book were intended to reinforce the notion that the BOXES method can be adapted to systems other than the trolley and pole. In each of these applications there is a different trigger that moves the system from reactive to reflective mode. In a continuous system, a timeout or sampling event allows the algorithm an opportunity to update its signature table values. In the disc access system, some preset time interval enables the update to the cache memory control unit to be activated. In the control of the complex trajectory of a chaotic system, a maximum runtime is imposed which is similar to the trolley and pole failure mechanism.

The BOXES system can be adapted to fit many systems but is hardly reaching the Bloom level 3 (application) as each version of the systems requires substantial intervention by a programer.

14.5.1 Chapter 8

This chapter described how the BOXES methodology can be adapted to control long duration continuous processes that may or may not be ill defined. By altering the rules of how the BOXES game is implemented, the system used timeouts as opportunities for positive statistical reinforcement and consequent improved performance. It is customary to expect that control systems that include elements of learning may frequently underperform or even fail in their initial stages of maturity. Good and bad control actions must be identified and cataloged so that strategies for avoiding, or at least postponing, losses (failures) can be formulated. However, when the system under control is continuous, for example, in process control, the system does not have the luxury of failure and restart as in the trolley and pole application.

The idea of using the BOXES method as a secondary rather than primary control mechanism is introduced. In support of this, an example BOXES augmentation of a PID tuning agent is included along with some preliminary results which indicated that quite drastic reduction in system oscillations can be expected using this refinement.

14.5.2 Chapter 9

Chapters 5 and 6 illustrated that the BOXES methodology works very well in prolonging the duration of motion of a freely hinged pole that is attached to a constantly moving cart driven by a theoretically instantly reversible motor. The algorithm reads the state variables and issues one of two possible control signals connected to the motor control system thus enforcing right/left motion. There are many other dynamic systems that are controlled in a similar bang-bang fashion and consequently candidates for the BOXES methodology.

This chapter described three case studies focusing on applications in which the BOXES system learns to switch a process variable on or off based on a signature table that is populated by appropriate learning laws. The systems described are a fedbatch bioreactor, a trash to steam plant, and a system tasked with reversing a tractor trailer into a restricted parking space.

14.5.3 Chapter 10

This chapter illustrated that the BOXES methodology can be adapted to assist in other rather esoteric non linear application areas. Two very different examples are included; the first relates to improving a time forecasting algorithm in a database disc file access system, and the second to reducing oscillations in a dynamic chaotic system. While the original BOXES algorithm needed some modification, the overarching principles were shown to remain intact.

Adaptive BOXES is an intelligent accessing methodology that is transparent to the user and designed to provide measurable performance upgrade using the familiar signature table in a novel way as it reveals disc caching opportunities. In the second application that illustrated the possible control of a chaotic system, the BOXES method was adapted to modify the coefficients in the well-known Lorentz equations with some startling results using an almost identical technique to that proposed to be used in augmenting a PID loop in Chap. 8.

14.6 Part IV: Improving the Algorithm

There have been many revisions and modifications made to the BOXES algorithm too numerous to mention. Chapters 11, 12 and 13 of the book showed possible alterations that are all designed to shorten the learning time before the automaton can be trusted to perform adequately. It may be argued that the Bloom levels 5 (synthesis) and 6 (evaluation) are in effect because of these modifications.

14.6.1 Chapter 11

Early chapters in the book showed that the BOXES algorithm does indeed learn to control systems such as the trolley and pole, but the slow erratic learning curve and the potential dependency on initial values in the signature table were always a matter of concern. Always being cognizant of the black box constraint that no a priori knowledge can be assumed, Chap. 11 proposed several modifications to the standard BOXES methodology, some of which were considered favorable in advanced system designs.

Apparently violating the black box prohibition, the first improvement to the system assigned commonsense unalterable values to certain regions in the signature table particularly those near a boundary condition such as the end stop on the trolley's track. This ensured that otherwise good runs that casually approach an extreme region of the solution space are not spoiled by a poor decision that could send the trolley careening off the track.

A second method described an algorithmic change designed to smooth out the influence of extreme run values. This statistical method only considers runs that are somewhat similar to current performance metric values, and disregards outliers. Thirdly, the idea of using cooperative neighbor states to enhance or even overturn a control decision is introduced. These advisor states are also rewarded or punished based on the quality of their contributions to the overall system performance. This is greatly expanded on in Chap. 12.

14.6.2 Chapter 12

Chapter Chapter 11 suggested one method to improve the rate of learning of the BOXES algorithm was to incorporate information from nearby states as possible modifiers of the current signature table value. This chapter described such a structural alteration to the methodology so that advisor cells are involved heavily in the real-time control decision. An advisor cell was defined as a cell that was positioned close to the current natural cell in the state space. For any particular cell, there can be a considerable number of advisors that vary only in one of the state integer values that were used to formulate its index. Data is available for each advisor that can provide its decision value (LEFT/RIGHT), a measure of the certainty of that decision, and an experience factor based on how often the advisor cell has been entered. Before a control decision was returned to the system under control, all possible advisors were consulted, and each was allowed to provide some level of advice. If the cumulative value of the *advice* of all qualified advisors outweighed the strength of the natural cell, an advisor-enhanced decision was asserted.

A simple advisor strategy was to adopt a voting scheme in which the *winner takes all* regardless of the certainty with which every control value was presented. To ensure the primacy of the natural cell is maintained, it can be given more than one vote in the aggregation. A better strategy was to estimate the strength, experience, and certainty of the values being offered as advice. If the net advice of the advisor cells is determined to be weak compared with the selected cell, it is ignored and the advisors held harmless. If there is strong contrary advice, the opposite value would be asserted.

In both cases, after a run is over, any advisor cells that contributed to the decision process are statistically rewarded or penalized based on the number and quality of their contributions. To illustrate the modification results obtained from simulations of an inverted pendulum was included in the chapter to illustrate the profound impact that advisor logic has on the learning curve of the system.

14.6.3 Chapter 13

The BOXES Methodology uses a signature table that is modified in real-time so that cells that the algorithm judges to have contributed toward control decisions are rewarded and punished in much the same way as a board game is played. Statistics are maintained over time producing what is equivalent to a knowledge base for each cell in the state space. These cells are identified by a unique system integer comprised of a combination of state integers.

Each state variable is divided into regions and processed as subsets of integers. The boundaries that define exactly where a state variable migrates from one region to another in previous versions of the software have always remained fixed in much the same way that any particular version of a game board is defined.

A possible improvement to ameliorate the latent disadvantages of the arbitrary selection of boundary locations was suggested in this chapter that used an evolutionary algorithm to modify these boundary positions. This is equivalent to redrawing a game board using a different aspect ratio. While the number of cells does not change, the statistical values of the individual cell strengths inside the signature table are used to alter the boundary locations inside the system and consequently reduce their initial arbitrary placements and improve the overall rate of learning.

14.7 Research Questions for Future Study

There are many unanswered questions in the BOXES methodology to which researchers across the globe have sought answers. This has led to many versions and variations of the BOXES software. Some have used the signature table data to produce crisp or fuzzy rules, others have reduced the number of possible states, while others have increased it, but there are some interesting questions that will need to be addressed. Three questions are broached below, but there are many more left unspoken in this volume and the interested researcher is encouraged to follow up on them.

14.7.1 Is There an Optimal Number of States?

When approaching any new application of the methodology, or after many runs are not producing much improvement in the performance of the system, it may be conjectured that there is some optimal number of states. Michie found that the familiar [$5 \times 3 \times 5 \times 3$] configuration that produces 225 possible states worked very well for a model of the trolley and pole. Implied in this configuration is the fact that the trolley position and pole angle require five possible states, which in fuzzy logic terms for the trolley position would be equivalent to [FAR_LEFT, LEFT, CENTRAL, RIGHT, and FAR_RIGHT]. The velocities of the trolley and the pole are only assigned one of three possible values each, with one always being the region near zero. But other options are possible. Should a system with fewer state variables require fewer state subdivisions? Conversely, should a system with many state variables need many more subdivisions? The consequence of adding signature table cells is that the software agent requires much more memory as the number grows. This is not a serious limitation to the software engineer especially if the computer upon which the software is to be hosted has a compiler that allows variable length array structures.

14.7.2 Is There a Generic Merit Formula?

So far, whenever a different application is envisioned, a different version of the merit calculation is needed that is usually very specific to the overall goal of the system. It is obvious that computing a time to failure is very different from reducing the area under a curve as a measure of computing bounce in a PID controller. Is it possible that some global function may exist that is application independent?

14.7.3 Is There a Generic Cell Strength Formula?

In this book, whenever a different application is discussed it has been possible to create some form of metric that conveys the notion of achievement of a goal. This might be an average runtime to be maximized, or a percentage bounce factor that is to be minimized. The fractal information dimension used in the Lorentz chaos example might prove useful in every application as it quantifies entrance and exit excursions from a unique state space cell. This could imply that every system regardless of its specific details improves as its dynamic complexity reduces.

14.8 Conclusions

The control of poorly defined, unstable, or heavily nonlinear dynamic systems has always been problematic using standard state space control paradigms. Control systems that incorporate imbedded artificial intelligence as a possible source of solutions are attractive in these cases because of their ability to tolerate uncertainty and make adjustments over time. Such learning systems can be trained to achieve better control performance using software that is designed to have the ability to adapt judiciously the distribution of control parameters. These parameters might be an amplifier gain, an on/off switch, or a feedback scaling value.

Shallow logic games that rely entirely on the selection of a board position such as tic-tac-toe and Connect 4® can be played in this way. At the end of each game a reward and penalty algorithm updates some value measure for each move based on the favorability of the outcome of the game. The values associated with particular board positions are then used in future play. When each game is over the values in the payoff matrix are updated based on the outcome of the game. Over many games in which the signature table algorithm wins and loses the payoff table becomes experienced in playing the game and may become a worthy opponent.

In more sophisticated games, such as chess or GO [7], it is not simply choosing the position to which a token should be moved, but rather which piece to move, what is its value, where it is now, and where might it best move to legally that is at stake. Such deep logic games are played by accessing logic trees and producing an array of possible move options and best selected within some allotted time span. In the seemingly simple game of GO, Kroeker [8] reports that despite using a method called the Monte Carlo Tree Search algorithm, the sheer number of play options is so large that even the Huygens supercomputer [9] running at 1,000X faster than IBM's Deep Blue has never defeated a professional GO player on a 19 × 19 board. In that same article, the author makes a very telling observation that the same game-playing algorithm is being configured to handle power plant management situations. In this application, the author reports that a failure in the plant is equivalent to an opponent's move in the game. This is exactly what the BOXES method seeks to do.

14.9 Some Final Thoughts

The BOXES methodology learns to adjust and implant profound control decisions into a signature table so that an ill-defined dynamic system can access them for use in system control. The method surprisingly needs little a priori knowledge of the system under control beyond the limiting values of the state variables and how each should be divided into integer regions. Unlike most control schemes, the mathematics of the system does not feature in the control algorithm.

This book is not intended as an all inclusive text on the BOXES method; it is a monograph of the author's adventures with the algorithm and an account of some of the modifications to the method and plausible application domains.

References

1. Doyle, J. 1991. The Foundations of Psychology: A Logico-Computational Inquiry into the Concept of Mind. In *Philosophy and AI: Essays at the Interface*, eds. Cummins, R & Pollock, J. Cambridge: MIT Press. pp. 65.
2. Baron, S. 1984. Adaptive Behavior in Manual Control and the Optimal Control Model. *Adaptive Control of Ill-Defined Systems*, eds. Selfridge, Rissland & Arbib, Plenum Press. 51–73.
3. Arthur, BW (1994) Inductive Reasoning and Bounded Rationality. *Amer. Econ. Review* 84: 406.
4. Chess openings and defences. http://www.eudesign.com/chessops/ch-group.htm (accessed August 22, 2011)
5. Rose S. *The Making of Memory: from molecules to mind.* Bantam Press, London: 103.
6. Bloom B (ed) et al. (1956).The Taxonomy of Educational Objectives in *The Classification of Educational Goals,* Handbook I*: Cognitive Domain.* New York: Longman.
7. Rules of GO http://en.wikipedia.org/wiki/Rules_of_Go#Board [Accessed August 15, 2011].
8. Kroeker KL 2011 A new benchmark for Artificial Intelligence. *Communications of the ACM.* 54:8 13–15.
9. http://huygens.supercomputer.nl/

Appendix A
Glossary of Terms and Abbreviations

Abbreviation

ACO	Ant colony optimization
ADC	Analog to digital convertor
AI	Artificial intelligence
DAC	Digital to analog convertor
DDC	Direct digital control
DSP	Digital signal processing
GPS	Global positioning system
IO	Input output
IT	Information technology
MENACE	Machine educable noughts and crosses engine
NLP	Natural language processing
PDA	Personal digital assistant
PID	Proportional, derivative and integral
SRI	Stanford Research Institute
VR	Virtual reality

Appendix B
Boxes Software Notes

Abstract Throughout the book, interested readers have been encouraged to write their own version of the software, and in some chapters software structures and logic were suggested. This appendix is a series of software notes designed to assist the researcher build working systems and experiment using the original BOXES method as a foundation. It is not a software engineering manual by any means but rather a set of practical suggestions that should prove useful. In order to create a working version of the software it is assumed that, while the actual programming language is not really that important, the programmer should have some reasonable level of experience in a procedural language such as C, Pascal, or Quickbasic (Noggle in QuickBASIC Programming for Scientists and Engineers. CRC Press, Boca Raton (1992); Cooper in Microsoft Quickbasic for Scientists: A Guide to Writing Better Programs. Wiley, (1988); http://en.wikipedia.org/wiki/QuickBASIC). For clarity and simplicity, code segments and data structures in this appendix are all based on the Microsoft® Quickbasic 4.5 compiler which are easily translated into other languages.

Disclaimer This appendix is purely a guideline for the interested researcher and no responsibility for its use or misuse is assumed by the author or publisher.

B.1 Quickbasic

Any procedural language compiler is acceptable for the creation of a working system. Quickbasic 4.5 is available as freeware and includes several libraries that must be linked in for correct operation. Programmers who are unaware of the features and limitations of this compiler should check the references [1–3] carefully. It is well known that Quickbasic was not written for the Windows© environment and many of the graphical niceties that programmers are accustomed to may have to be foregone or custom assembler language code written inside the Windows Command Processor which is easily linked to the application.

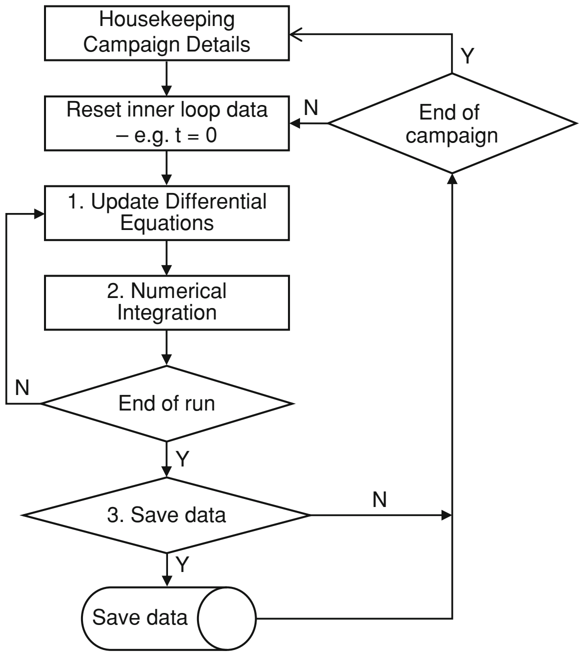


Fig. B.1 Simulation software loop

B.2 Essentials of Simulation Software

Writing code to simulate a dynamic system is relatively simple once the concept of numerical integration and acceptable accuracy are understood. Writing software that must execute in real-time is another matter and will be briefly touched upon in Sect. B.3. The simulation loop, shown in Fig. B.1 includes several essential aspects of writing programs for simulated system research.

Of particular interest is the fact that many millions of loops may need executing, so always having an animated graphic of the system, for example a trolley and pole, in view is infeasible and of little import.

To assist the coder, in the first stages of the simulation, the simulation loop can be redrawn according to Fig. B.2 to show how a modular structure can be asserted into the design using subprograms.

Each subprogram in Fig. B.2 is listed in Table B.1 and described in the given referenced sections which form the basis for a software model for the bOXES algorithm. In addition, Sect. B.2.4 illustrates the software required for the inner loop.

This software framework can be used for any simulated system in any program designed to represent a numerical solution to a set of differential equations be they linear or otherwise in which the BOXES control method has been selected. The only section that relates to the system under control is the MODEL subroutine.

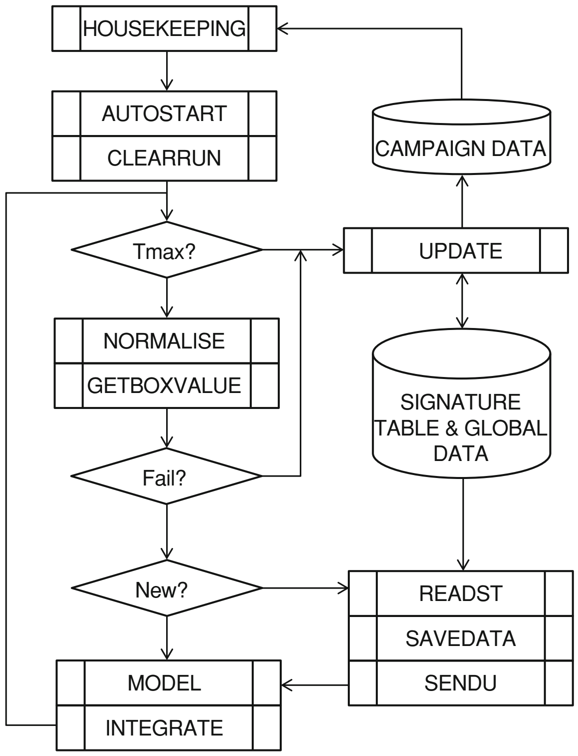


Fig. B.2 Simulation loop subprograms

Table B.1 Initial BOXES software subprograms

Subprogram	Function	Reference
HOUSEKEEPING	Defines all fixed data and shared variable space	B 2.1
AUTOSTART	Sets all state variables with valid (random) initial conditions	B 2.2
CLEARRUN	Asserts and clears any data fields that are used during a control run	B 2.3
NORMALISE	Processes real state variables into a set of normal (0 ..1) values	B 2.5
GETBOXVALUE	Translates normalized variables into integer zones and creates the system integer	B 2.6
READST	Using the state integer, the current value of the signature table is read	B 2.7
SAVEDATA	If the state has changed, in-run data for use in the UPDATE phase are saved	B 2.8
SENDU	Control value (u) passed back to loop—and may be modified by advisors	B 2.9
MODEL	Using data from SEND u the system's differential equations are defined	B 2.10
INTEGRATE	Differential equations are all numerically integrated and the time advanced.	B 2.11
UPDATE	Data from the run just ended is merged into global data defined by merit	B 2.12

B.2.1 Housekeeping

This is the subprogram HOUSEKEEPING in which the software designer must be careful to select data structures for the many variables needed for a successful BOXES implementation. Quickbasic allows the use of global variables but at the cost of program memory which can be exceeded in run time. Like other programming languages, Quickbasic supports a subroutine in which real memory is assigned only when that particular code segment is executed. For the present, the data structures to be demonstrated in this section relate to the four state variable system used in prior chapters of the book for the trolley and pole.

It is suggested that the following structures be used as global variables as they are constantly accessed or updated during the simulation run. Quickbasic uses the DIM (dimension) statement to declare variables that are arrays. For the trolley and pole configuration used throughout the previous chapters there are 225 cells or BOXES; this determines the array sizes.

In run cell data for BOXES program for LEFT and RIGHT token values

```
DIM LTIMIN (225), LCOUNTIN (225)
DIM RTIMIN (225), RCOUNTIN (225)
```

Signature table array

```
DIM PHI (225)
```

Where, for each cell value M

LTIMIN ()	= time spent inside cell during when decision is LEFT
LCOUNTIN ()	= number of times cell is entered with decision = LEFT
RTIMIN ()	= time spent inside cell when decision is RIGHT
RCOUNTIN ()	= number of times cell is entered with decision = RIGHT

PHI () = BOXES signature table array

State variables and simulated time

```
X, DX, THETA, DTHETA, D2X, D2THETA
```

Where,

X	= trolley position.	THETA	= pole angle
DX	= velocity of trolley	DTHETA	= pole velocity
D2X	= acceleration of trolley	D2THETA	= pole acceleration

Control variables

M = current system integer value
u = +1 or −1 in signature table PHI(m)
Mold = system integer in last iteration

Simulated time

```
TIME, DELTAT, TFAIL, TMAX
```

Where,

TIME	= duration of run	DELTAT	= integration step
TFAIL	= Time when run fails	TMAX	= Max run time allowed

B.2.2 Autostart

Subroutine AUTOSTART asserts reasonable initial values for the four state variables according to Chap. 7. In an early simulation it is probably easier to assign a random number somewhere within 20–80% of the range for each normalized variable. Using the system time as the seed of the random number generator ensures that the initial values created are as close to random as possible. There are other ways to seed the random numbers and the programmer is left to explore these.

Random number seed reset to system time

RANDOMISE (time)

Initial values between 0.2 and 0.8

X_{norm} = 0.2 + RND(0) * 0.6
DX_{norm} =0.2 + RND(0) * 0.6
TH_{norm} = 0.2 + RND(0) * 0.6
DTH_{norm} = 0.2 + RND(0)* 0.6

Translation back to real state variables

$X = X_{min} + (X_{max} - X_{min})*X_{norm}$
$DX = DX_{min} + (DX_{max} - DX_{min})*DX_{norm}$
$TH = TH_{min} + (TH_{max} - TH_{min})*TH_{norm}$
$DTH = DTH_{min} + (DTH_{max} - DTH_{min})*DTH_{norm}$

B.2.3 Reset Inner Loop Variables

This is the subprogram CLEARRUN that at the beginning of each simulation run clears the run-dependant variables to their initial values. For example, the identifier *time* that represents the duration of the current run must be set to zero and any BOXES data that will be stored for use in the update phase must also be cleared.
Initialize time and cell number

MOLD = 0: TIME = 0

Clear in-run cell data

```
FOR I = 1, 225
    LTIMIN(I) = 0
    LCOUNTIN(I) = 0
    RTIMIN(I) = 0
    RCOUNTIN(I)=0
NEXT I
```

B.2.4 Inner Loop Control

The simulation program will execute until some maximum time is exceeded or until the system fails. In early stage testing, the value of this maximum time can be set quite small so that test data can be inspected. If a failure occurs, which in the trolley and pole system occurs if the value of one of the state variables becomes out-of-range, the value of the system integer (the box number) is set to zero. A code structure in the MAIN program such as the following controls this program campaign.

```
DO
    << See Fig. B.2 for the inner loop processes >>
UNTIL T>TMAX OR UNTIL m = 0
    << Begin UPDATE process and next run sequence >>
```

B.2.5 Normalization of State Variables

In the NORMALISE code segment, the state variables are processed to return values in the zero to one range. To accomplish this, minimum and maximum values of each state variable must be known. Critics of the method assert that this is a priori knowledge that would disqualify BOXES from being a black box method. However, it is not unusual for a system designer to know these control limits from prior systems or from observing the system, so they can be classified rather as common system knowledge.

The normalization process

$$Ynormal = (Y - Y\text{min})/(Y\text{max} - Y\text{min})$$

Where,

Y	= Any state variable X, DX, THETA, DTHETA
Ymax	= Known maximum value of that variable
Ymin	= Known minimum value of that variable

The values of these upper and lower limits may be dictated by physical constraints, safety values, or common sense.

B.2.6 Calculation of System Integers

Once the state variables are normalized, each one can now be processed to give individual values of the state integers, which will be combined into the system integer or box cell value. This subprocess is called GETBOXVALUE in Fig. B.2. This routine requires boundary values for each state normalized variable based on the number of boxes to be configured. For the usual $5\times3\times5\times3$ configuration, the boundary arrays would be for the trolley and pole variables:

Boundary arrays

```
DIM BX(5), BDX(3), BTH(5), BDTH(3)
```

End values

The end values of the boundaries are 0 and 1 because all normalized variables now lie within this range.

```
BX(0) = 0: BDX(0) = 0:BTH(0) = 0:BDTH(0) = 0
BX(5) = 1: BDX(3) = 1:BTH(5) = 1:BDTH(3) = 1
```

Example boundaries—note index 0 and 5 are the limiting values zero and 1

```
BX(0)=0: BX(1)=0.2 :BX(2)=0.4: BX(3)=0.6: BX(4)=0.8: BX(5)=1
```

Default values

```
M = 0: zoneX = 0 etc.
```

Compute state integer—for example for variable X.

```
FOR zone=0 TO 4
  IF x > BX(zone) AND x < BX(zone+1) THEN zoneX= zone +1: EXIT
NEXT zone
```

Example for X=0.5 for the given boundaries

Table B.2 shows how this software executes for X=0.5 and shows how the value of zone=3 is obtained.

Figure B.3 is a further illustration of the process.

Table B.2 Execution of loop

Loop integer "zone"	BX(zone)	BX(zone+1)	State integer "i" value
0	0	0.2	i =0
1	0.2	0.4	i =0
2	0.4	0.6	i =3
EXITS LOOP			

	X = 0.5 •					
Boundary	• 0	• 0.2	• 0.4	• 0.6	• 0.8	• 1.0
FAIL	zone=1	zone=2	zone=3	zone=4	zone=5	FAIL
State integer zoneX = 3 •						

Fig. B.3 Example zone location for x=0.5

Compute system integer

Once all four state integers have been identified, it is simple to combine them according to Sect. 5.2.5 using code such as:

```
M= zoneX +5*(zoneDX - 1) + 15*(zoneTH - 1) + 75*(zone DTH - 1)
```

Did a failure occur?

Should any state integer not lie within the zero to one range, the zone value for that variable is set to zero and the system integer set to zero accordingly by a simple multiplication:

```
MTEST = zoneX*zoneDX*zoneTH*zoneDTH
IF MTEST = 0 THEN M=0
IF M = 0 THEN TFAIL=T: EXIT
```

B.2.7 Access Signature Table

The new value of system integer (*m*) is used as an index for the natural value of the signature table (*PHI*) and used as a simple array pointer inside the routine READST.

Access signature table

```
u = PHI(m)
```

The current value of signature table for this value of *m* is either plus or minus 1. This equates to a LEFT/RIGHT decision that is passed on to the motor control center.

B.2.8 Saving In-Run Data

The routine SAVEDATA records times and counts as they happen during the run whenever the cell number changes which means that the system variables have migrated into other regions of operation under the driving force of the trolley motor. Only if a change in state has occurred should data be updated. In this code section, data is collected for the state in which the system currently lies and updates the local arrays *TIMIN(m)* and *COUNTIN(m)* based on the trajectory of the solution as represented by the simulated time and progression of the algorithm between the states.

```
Check if value of m altered since last sample—if not ignore

IF M = MOLD EXIT

New value of M—so update appropriate sum of entry time and count

IF U=RIGHT THEN
update right side data

            RCOUNTIN(m)=RCOUNTIN(M) + 1
            RTIMIN (M)= RTIMIN(M) + TIME
          ELSE

update left side data

            LCOUNTIN(m)=LCOUNTIN(M) + 1
            LTIMIN (M)= LTIMIN(M) + TIME
          ENDIF

Save the new m value for next entry

            MOLD=M
```

B.2.9 Attaching the BOXES Controller

Initially, the subprogram SENDU simply passes the natural control value from the signature table to the model of the system under control. In more advanced software systems, this is where any advisor logic and subsequent revision of the control value can be implanted.

B.2.10 State Equations of the System Model

In this routine, MODEL, the model to be simulated is encoded into the differential equations that describe each state variable. The control value that BOXES provides is represented by the variable *u* and is supplied in the previous section. Equations 5.1 and 5.2 can be encoded in several ways, for example:

```
D2X part1= u*F – Mp*L*DTH*COS(THETA)
D2Xpart2 = Mp*L*(DTHETA)*(DTHETA)*SIN(THETA)
D2X = (D2xpart1+D2Xpart2) / (Mt+Mp)
I =Mp*L/3
D2THETA = (Mp*L*G*SIN(THETA) – Mp*L*D2X*COS(THETA))
D2THETA= D2THETA /(I + Mp*L*L)
```

Where,

Mt = Mass of trolley	Mp = Mass of pole
F = Motor force	L = ½ Length of pole
I = Inertia of pole	u = BOXES control variable
G = Acceleration due to gravity	

B.2.11 Numerical Integration

Using the Euler method, or any other numerical integration algorithm the values of all variables that are expressed by the differential equations can be advanced over the finite time interval (DELTAT) in the section designated INTEGRATE in Fig. B.2. The Euler method is simplest to implement as it simply extends variables using their derivatives over a short time in which the derivative is assumed not to have changed by a significant amount. The value of DELTAT, known as the integration step length, is somewhat arbitrary but does determine the accuracy of the simulation. If it is a very small value, the results are more accurate but the run time may become excessive. If a campaign has a million scheduled runs, this becomes a real problem.

Compute new values of the state variables using Euler method

```
NEWDX= DX + H*D2X
NEWX = X + H * DX
NEWDTHETA = DTHETA + DELTAT*D2THETA
NEWTHETA = THETA + DELTAT * DTHETA
```

Save new values into their identifiers

This is necessary to prevent newly computed values overriding incoming variables.

```
X= NEWX
DX=NEWDX
THETA=NEWTHETA
DTHETA = NEWDTHETA
```

Update simulated time by the integration step length

```
TIME = TIME + DELTAT
```

B.2.12 Saving Global Data at the End of a Run

The UPDATE procedure serves two purposes; at first it merges saved cell-by-cell historical data with data from the last run. Secondly, it saves a historical record of the progression of the system merit over many runs.

Run by run update

In this UPDATE routine, global and cell-by-cell data is merged with data from the last run and any necessary alterations to the signature table data values made. In its elemental stages, the BOXES database contains the following data structures

```
GLOBAL_LIFE, GLOBAL_COUNT
DIM LLIFE(225), LCOUNT (225), RLIFE(225), RCOUNT(225)
```

Where,

GLOBAL_LIFE = total weighted run time of the system so far
GLOBAL_COUNT = total weighted experience of the system so far
LLIFE() = Total time measure for each cell when its value is “L”
LCOUNT() = Total entry count for each cell when its value is “L”
RLIFE() = Total time measure for each cell when its value is “R”
RCOUNT() = Total entry count for each cell when its value is “R”

Update system merit and desired level of achievement

At the end of each run, the global values are aged and updated.

```
GLOBAL_LIFE=GLOBAL_LIFE*$\delta$k + $T_F$
GLOBAL_COUNT=GLOBAL_COUNT *$\delta$ k +1
MERIT = GLOBAL_LIFE/GLOBAL_COUNT
DLA = $C_0$ + $C_1$ * MERIT
```

Where,

δk = Aging factor (e.g. 0.99)
T_F = Time to fail of last run

DLA = Desired level of achievement
C_0, C_1 = Learning coefficients

Update historical cell by cell data

In this phase of the update operation, data saved using the SAVEDATA routine is added to the weighted historical data based on the current data value corresponding to the cell number in the signature table. Each cell is processed regardless of if they contributed to the performance of the last run.

Age all historical data

```
FOR M=1 TO 225
LLIFE(M)=LLIFE(M)* $\delta$k: LCOUNT(M)=LCOUNT(M)* $\delta$k
RLIFE(M)=RLIFE(M)* $\delta$k : RCOUNT(M)=RCOUNT(M)* $\delta$k
NEXT M
```

Process all cells again

```
FOR M=1 TO 225
```

Get current signature table value

```
U=PHI(m)
IF U = "L" THEN
            LLIFE(M) = LLIFE(M)+ LTIMIN(M)
            LCOUNT(M)=LCOUNT(M)* + LCOUNTIN(M)
        ELSE
            RLIFE(M) = RLIFE(M) + RTIMIN(M)
            RCOUNT(M)=RCOUNT(M)* + RCOUNTIN(M)
        ENDIF
```

Compute cell strengths for L and R values using DLA and the learning rate K (see Eqs. 5.14–5.16)

```
LST = (LLIFE(M) + K* DLA)/(LCOUNT(M) +K)
RST = (RLIFE(M) + K*DLA)/(RCOUNT(M) + K)
```

Update signature table

```
IF LST > RST THEN PHI(M)= "L"
IF RST>LST THEN PHI(M)= "R"
```

This method does not update the signature table (PHI) if both "L" and "R" data show equal strength. Other researchers randomly reset the signature table.

```
NEXT M
```

Historical archiving

The second process, after the signature table and historical data processing are completed is to record what Fig. B.2 designates as CAMPAIGN DATA which is a permanent record of the progress of learning over many runs. In high data volume cases, it is critical that some metric, such as global merit, are computed regularly and stored on permanent removable media such as a thumb-drive. The data that is recorded in this simple fashion is permanent and remains after a computer failure or loss of power. Quickbasic has a feature that allows sequential records to be added to a sequential file and allows the logical unit to be closed, and therefore removed, between updates.

Save campaign data every n^th^ run

One simple algorithm for this is to maintain a *savecount* integer that is incremented upon entry into the UPDATE process. When this reaches a certain value, *savenow*, which is present before the campaign begins, a data record is written.

Increment counter

```
SAVECOUNT = SAVECOUNT + 1
```

Check for save event request

```
IF SAVECOUNT = SAVENOW THEN
        WRITE ARCHIVE record
        SAVECOUNT=0
ENDIF
```

Campaign data record

The programmer may elect to save the current run number and merit in this simple record structure along with any other test or debug data that is deemed necessary. By using a simple text file, results can be ported easily into Excel or WORD for further processing and presentation.

B.3 Real-Time Software

When the BOXES method is to be applied to a real system, such as a physical trolley and pole, the software designer should be aware that the task is many times more difficult. Experience has proven that the collection of date using purchased firmware imposed certain restrictions on how effective the BOXES method can be. The system time constant may not be the same as that necessary for BOXES data processing during a run so great care must be taken. In the Liverpool system custom hardware was written that interfaced into assembler language routines to ensure that all timing, logic, and accuracy considerations were met. This requires certain knowledge regarding the real-time operating system being used and how it processes event flags and handles shared memory. Having stated this, the avid researcher is encouraged to implement the BOXES algorithms in real-time using whatever means are available.

B.4 Conclusion

The BOXES algorithm uses much of the above methodology and it is only the definition of the "board", the "tokens", "the rules", and the "strategies" that differ from implementation to implementation. Rather than winning a game, in the real-time control application, the object is rather to aggressively learn to prolong the game so the system remains in control. In the case of the trolley and pole, the trolley moves back and forth, preferably across the middle of the track, and the pole swings in concert about the vertical. Data associated with each system integer is aged over time so that past performance is progressively forgotten, and contributing states are rewarded or penalized based on the outcome of each control run.

Because the number of individual runs can be quite large, careful design of the overall campaign must be made prior to the start of a sequence of runs. No one control run is retrievable or important in itself, it is the increase in the overall merit of the system over time that is the goal of the software. Snapshot values of the system data can be designed by saving every hundred runs or so.

Programmers should keep with whatever language with which they are most familiar as the BOXES method is not algorithmically difficult.

References

1. Noggle, J.H (1992) QuickBASIC programming for scientists and engineers. CRC Press, Boca Raton FL. ISBN 0-8493-4434-4
2. Cooper J.W. (1988) Microsoft Quickbasic for Scientists: A Guide to Writing Better Programs. John Wiley & Sons. 0-4716-1301-0
3. http://en.wikipedia.org/wiki/QuickBASIC

Appendix C
BOXES Publications, Lectures, and Presentations by the Author

Abstract The following is a date ordered list of presented and published articles by the author. Many of these were part of invited, keynote or conference lectures where input from scholars on a worldwide basis was invited and enjoyed. Many researchers have worked with a trolley and pole and have found that it is fairly easy to simulate its dynamic performance but difficult to realize in practice. Many control strategies have been attached to its motor control and it has been the intent of this monograph to show how the BOXES method adds another level of richness to intelligent control. Many of the details inside this book came directly or indirectly from these works and grateful acknowledgement is made to them.

C.1 Reference List of Publications

1. Russell, D.W. An Evolutionary Machine Learning Algorithm for Real-time Dynamic Systems *–Proc. 2011 African Conference on Software Engineering and Applied Computing* ACSEAC, Cape Town South Africa.
2. Russell D.W. Adaptations of the BOXES Algorithm for Improved Black Box Control of Dynamically Unstable Systems – *Proc.* MX 2006 *10thInternational Mechatronics International Conference,* Malvern June 2006.
3. Russell D.W. A Self-Organizing Real-Time Controller. *Institute for Computational Science*, Spring Computational Science Invited Seminar Series; University Park April 24, 2006.
4. Russell, D.W. Control of Dynamically Unstable Systems Using a Self Organizing Signature Table Controller. *Invited Lecture*. October 6, 2005 University of Limerick, Ireland.
5. Russell, D.W. On the Control of Dynamically Unstable Systems Using a Self Organizing Black Box Controller. *Proc 7thBiennial ASMA Conference on Engineering Systems Design and Analysis- ESDA 2004*, Manchester, UK, July 19-22, 2004. CD

6. Russell, D. W. The Control of Dynamically Unstable Systems using a Black Box Controller. *Invited Colloquium*, Penn State University Park, November 20, 2003.
7. Russell D.W., A Proposed Evolutionary, Self-Organizing Automaton for the Control of Dynamic Systems. *Proc. RSCTC 2002 3rdInternational Conference on Rough Sets and Trends in Computing*. October Malvern PA, 2002 pp. 33-43.
8. Alpigini J.J. & Russell, D.W., System Analysis via Performance Maps. *Control EngineeringPractice,* Vol. 11, No. 5, 2003 Elsevier Press. pp. 493-504.
9. Alpigini, J.J., and Russell, D.W., The Effective Information Dimension: A Metric for Complexity Measurement, *Proc. of the IASTED International Conference on Applied Simulation and Modeling (ASM 2000)*, Banff, Alberta, Canada, July 24-26, 2000,162-167.
10. Russell, D.W. & Wadsworth, D. L., Database Retrieval Enhancement Using the BOXES Methodology. *Proc. IFAC Workshop: AIRTC'98*: *AI in Real Time Control.* Grand Canyon, AZ, October 5-8, 1998.
11. Russell D.W. Dynamic Systems and Chaos *IFAC Workshop: Proc. AIRTC'98: AI in Real Time Control.* Grand Canyon, AZ, October 5-8, 1998 (Invited Tutorial).
12. Alpigini, J.A. and Russell, D.W., Visualization of Control regions for Badly-Behaved Real-Time Systems. *Proc. IASTED Intl. Conf. On Modeling & Simulation, P*ittsburgh, PA, May 13-16, 1998, 271-276.
13. Russell, D.W. and Alpigini, J.A. Visualization of Controllable Regions in Real-Time Systems using a Julia Set Methodology. *Proc. IEEE Intl. Conf on Information Visualization IV'97*, London, UK Aug 27-29,1997 (Keynote) 25-31.
14. Russell, D.W., The Occurrence and Control of Chaos in Dynamic Systems. *Short Course ERASMUS Program* –Karlsruhe August 25 1995.
15. Russell, D.W., Advisor Enhanced Real–Time System Control. *Control Engineering Practice*. 1995 Vol. 7: No.3 1995 Pergamon Press, UK 977-983.
16. Russell, D.W., Using the BOXES Methodology as a Possible Stabilizer of Lorentz Chaos. *Proc. AI'94 7th.Australian Joint Conference on AI:* Armidale, NSW, Australia Nov. 21-25 1994 338-345.
17. Russell, D.W. Stabilization of Chaotic Systems. *IEEE TENCON'94*, Singapore, August 23-26 1994. (Invited 1/2 day tutorial).
18. Russell, D.W. Stabilization of Chaotic Systems using the BOXES Methodology. *Proc. 9th Intl. Conf. on Applications of Artificial Intelligence in Engineering AIENG'94*. July 19-21 1994 Malvern. PA, Computational Mechanics Institute 295-303.
19. Russell, D.W. Failure Driven Learning in the Control of Ill–defined Continuous Systems *Int. Journal of Cybernetics & Systems.* 1994: Vol. 25: Taylor & Francis. Washington DC: 555-566.
20. Russell, D.W. An Advisor enhanced Signature Table Controller for Real time, Ill defined Systems. *Proc. IFAC Symposium on Real Time Control SICICA'94*. Budapest, June 8-10 1994 66-71.

21. Russell, D.W. A Critical Assessment of the BOXES Paradigm. *Applied Artificial Intelligence – an International Journal*, Vienna. 1993: Vol. 7: No. 4 383-396.
22. Russell, D.W., SUPER–BOXES—An Accountable Advisor System. *Proc. AIENG'93 8th Intl. Conf. on Applications of AI in Engineering*, Toulouse, France. 29 June– 1 July 1993, Elsevier Press, 477-489.
23. Russell, D.W., Application of the BOXES Methodology to Continuous Systems. *Journal of Engineering Applications of Artificial Intelligence*, 1993: Vol. 6: No 2. Pergamon Press, UK 145-152.
24. Russell, D.W., Statistical Inferencing in a Heuristic Real Time Controller. *Proc. 4th Int. Workshop on AI and Statistics,* January 3–6 1993, Fort Lauderdale, FL, 357-362.
25. Russell, D.W., The BOXES Methodology – An Introduction. *AI Expert*, 1993: Vol. 8: No. 1. January, Miller Freeman, San Francisco, CA 22-27.
26. Russell, D.W., AI Adaptive Learning in Ill–defined Systems. *Proc.ICARCV'92, 2nd. Int. Conf. on Automation, Robotics & Computer Vision.*, Sept. 14–17 1992, Singapore. World Scientific Vol. 3: INV-4: 1-4 (Invited).
27. Russell, D.W. AI Augmented Goal Achievement. *Proc. AIENG'92, 7th. Int. Conference – Applications of AI in Eng.* July 14–17 1992, Waterloo, Canada, Elsevier Applied Science, London, 971–982.
28. Russell, D.W., AI Augmented Goal Achievement in Process Control Systems. *Proc. 11th European Mtg. on Cybernetics & Systems Research.* Vienna, April 21-24, 1992. World Scientific 1625-1632.
29. Russell, D.W., Further Studies in AI Augmented Process Control. *Proc. AIRTC'91, 3rd IFAC Intl. Workshop on Artificial Intelligence in Real Time Control.* Sonoma, CA, Sept. 23-25 1991, Pergamon Press, NY. (75-79.
30. Russell, D.W. AI Augmented Process Control using the BOXES Methodology. *Proc. AIENG'91: 6th Int. Conference – Applications of AI in Engineering,* Oxford University, UK. July 2-4 1991, Elsevier Applied Science, London, 611-622.
31. Russell, D.W. The Trolley and Pole Revisited: Further Studies in AI Control of a Mechanically Unstable System. P*roc. AIENG'90: 5th International Conference: Applications of AI in Engineering* Vol. II Springer-Verlag. 307-322.
32. Russell D.W. The Trolley and Pole Revisited – Invited lecture March 13, 1990. The Turing Institute, Glasgow Scotland.
33. Russell, D.W., S.J. Rees and Boyes, J.A. Self Organizing Automata. *Proc. 1977 Conference on Information and System Sciences* 1977, Johns Hopkins University, Baltimore, MD April 1977. 226-230.
34. Russell, D.W. and Rees, S.J. System Control: a case study of a Statistical Learning Automaton. *Progress in Cybernetics and Systems Research 1975:* Vol 2: and *Proc. 2nd European Meeting on Cybernetics & Systems Research.*1974: Vienna, Austria, Hemisphere Publishers 114-120.

Author Biography

David W. Russell, Ph.D, C.Eng, FBCS, FIET, FIMechE, is a Professor of Electrical Engineering at Penn State Great Valley, Pennsylvania, USA. His research interests include: applications of artificial intelligence (AI) to real-time, poorly defined control systems, systems engineering, factory information systems, philosophy of machine intelligence, and chaos theory and applications. He has lectured worldwide for many years and has over 30 papers based around the BOXES methodology.

Index