Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

EVOLUTIONARY POLYMORPHIC NEURAL NETWORKS IN CHEMICAL ENGINEERING MODELING

by
Li Gao

Evolutionary Polymorphic Neural Network (EPNN) is a novel approach to modeling chemical, biochemical and physical processes. This approach has its basis in modern artificial intelligence, especially neural networks and evolutionary computing. EPNN can perform networked symbolic regressions for input-output data, while providing information about both the structure and complexity of a process during its own evolution.

In this work three different processes are modeled: 1. A dynamic neutralization process. 2. An aqueous two-phase system. 3. Reduction of a biodegradation model. In all three cases, EPNN shows better or at least equal performances over published data than traditional thermodynamics/transport or neural network models. Furthermore, in those cases where traditional modeling parameters are difficult to determine, EPNN can be used as an auxiliary tool to produce equivalent empirical formulae for the target process.

- Feedback links in EPNN network can be formed through training (evolution) to perform multiple steps ahead predictions for dynamic nonlinear systems.

- Unlike existing applications combining neural networks and genetic algorithms, symbolic formulae can be extracted from EPNN modeling results for further theoretical analysis and process optimization.

- EPNN system can also be used for data prediction tuning. In which case, only a minimum number of initial system conditions need to be adjusted. Therefore,

the network structure of EPNN is more flexible and adaptable than traditional neural networks.

- Due to the polymorphic and evolutionary nature of the EPNN system, the initially randomized values of constants in EPNN networks will converge to the same or similar forms of functions in separate runs until the training process ends. The EPNN system is not sensitive to differences in initial values of the EPNN population. However, if there exists significant larger noise in one or more data sets in the whole data composition, the EPNN system will probably fail to converge to a satisfactory level of prediction on these data sets.
- EPNN networks with a relatively small number of neurons can achieve similar or better performance than both traditional thermodynamic and neural network models.

The developed EPNN approach provides alternative methods for efficiently modeling complex, dynamic or steady-state chemical processes. EPNN is capable of producing symbolic empirical formulae for chemical processes, regardless of whether or not traditional thermodynamic models are available or can be applied. The EPNN approach does overcome some of the limitations of traditional thermodynamic/transport models and traditional neural network models.

EVOLUTIONARY POLYMORPHIC NEURAL NETWORKS IN CHEMICAL ENGINEERING MODELING

by
Li Gao

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Chemical Engineering

Department of Chemical Engineering

August 2001

Copyright © 2001 by Li Gao

ALL RIGHTS RESERVED

APPROVAL PAGE

EVOLUTIONARY POLYMORPHIC NEURAL NETWORKS IN CHEMICAL ENGINEERING MODELING

Li Gao

Dr. Norman W. Loney, Dissertation Advisor — Date
Associate Professor of Chemical Engineering,
New Jersey Institute of Technology

Dr. Basil C. Baltzis, Committee Member — Date
Professor of Chemical Engineering, Acting Chair,
New Jersey Institute of Technology

Dr. Robert B. Barat, Committee Member — Date
Associate Professor of Chemical Engineering,
New Jersey Institute of Technology

Dr. Dana E. Knox, Committee Member — Date
Associate Professor of Chemical Engineering,
New Jersey Institute of Technology

Dr. Denis L. Blackmore, Committee Member — Date
Professor of Mathematics,
New Jersey Institute of Technology

Dr. Daniel J. Wasser, Committee Member — Date
Principal Application Developer,
Foster Wheeler Corporation

BIOGRAPHICAL SKETCH

Author: Li Gao

Degree: Doctor of Philosophy

Date: August 2001

Undergraduate and Graduate Education:

- Doctor of Philosophy in Chemical Engineering,
  New Jersey Institute of Technology, New Jersey, 2001
- Bachelor of Engineering in Chemical Engineering,
  Tsinghua University, Beijing, P.R.China, 1997
- Bachelor of Science in Computer Science and Engineering
  Tsinghua University, Beijing, P.R. China, 1997

Major: Chemical Engineering

Publications and Presentations:

Li Gao and Norman Loney,
"A New Mixed Neural Network Model for Prediction of Phase Equilibrium in Two-Phase Extraction System," *Industrial and Engineering Chemistry Research* (Submitted).

Li Gao and Norman Loney,
"Evolutionary Polymorphic Neural Network in Chemical Process Modelling," *Computers and Chemical Engineering* (In Press).

To my family and parents, for their unconditional support and love and constant encouragement.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my dissertation advisor and mentor, Dr. Norman Loney, whose kind enthusiasm for research in the field of modeling instilled in me the perseverance to finish my Ph.D. dissertation. I would also like to thank Dr. Basil Baltzis, Dr. Dana E. Knox, Dr. Denis L. Blackmore, Dr. Robert Barat, Dr. Robert Luo and Dr. Daniel Wasser who served in my thesis committee and provided me with valuable feedback on various aspects of my thesis. Also, I would like to express my special gratitude to Dr. Robert Barat for his participation in my thesis committee.

I would also like to acknowledge the financial support of the NJIT Department of Chemical Engineering, Chemistry and Environmental Science during the course of my graduate study. In addition, I would like to express my sincere gratitude to Dr. Norman Loney and other faculty members in the Department for their ongoing efforts in providing and improving the research environment and computing facilities during my dissertation research.

Last, but not the least, I would also like to thank my loving parents and my brother in China for their endless and unconditional patience, love, support and constant encouragement.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION 1

2 LITERATURE REVIEW 5

2.1 Overview of Traditional Methods 5

2.2 Overview of Artificial Intelligence 9

2.2.1 Artificial Neural Networks 10

2.2.2 Evolutionary Computing 15

2.2.3 Current Trends in Artificial Intelligence 20

2.3 Applications in Chemical Engineering 21

3 OBJECTIVES 24

4 EPNN MODELING 26

4.1 Introduction 26

4.2 General Model Description 27

4.2.1 System Structure 27

4.2.2 Modeling Algorithms 31

4.2.3 Major Modeling Parameters 34

4.2.4 Comparison with Other A.I. Techniques 35

4.3 Model Implementation 36

5 EXPERIMENTS AND RESULTS 44

5.1 Overview 44

5.2 Dynamic Chemical Reaction System 44

5.2.1 Introduction 44

5.2.2 Results and Discussion 47

5.3 Aqueous Two Phase Extraction System 52

5.3.1 Introduction 53

TABLE OF CONTENTS
(Continued)

Chapter	Page
5.3.2 Data Preprocessing	54
5.3.3 Results and Discussion	56
5.4 Bioremediation Model Reduction	66
5.4.1 Introduction	66
5.4.2 Data Processing and Results	68
5.4.3 Discussion	72
6 SUMMARY AND CONCLUSIONS	78
7 LIMITATIONS AND FUTURE WORK	81
APPENDIX A SOURCE CODES	84
APPENDIX B INTERNET RESOURCES	120
REFERENCES	122

LIST OF TABLES

Table **Page**

5.1 Comparisons between EPNN prediction and experimental data for PEG 400 in bottom phase 65

5.2 Comparisons between EPNN prediction and experimental data for PEG 400 in top phase 65

5.3 Comparisons between EPNN prediction and experimental data for PEG 1000 in bottom phase 65

5.4 Comparisons between EPNN prediction and experimental data for PEG 1000 in top phase 65

5.5 Prediction comparison for experiment K-13 75

LIST OF FIGURES

Figure		Page
2.1	Sketch of a biological neuron	11
2.2	A typical structure of artificial neuron	13
2.3	A sample feedforward neural network	14
2.4	A sample recurrent neural network	15
2.5	A typical evolutionary algorithm	17
2.6	An example of string representation of numbers in genetic algorithms	18
2.7	An example of single point mutatation in genetic algorithms	19
2.8	An example of single point crossover in genetic algorithms	19
2.9	An example of a tree representation of computing formula	20
4.1	A typical composition of EPNN modeling system	28
4.2	A sample EPNN individual network	29
4.3	Tree representation for the formula of node N_0 in Equation 4.2	30
4.4	A typical EPNN network training algorithm	39
4.5	Some typical operations during elementary mutation	40
4.6	A typical crossover operation in tree representation	41
4.7	A typical crossover operation at EPNN individual level	42
4.8	EPNN coding hierarchy scheme	43
5.1	Neutralization CSTR reaction	46
5.2	CSTR EPNN prediction results	48
5.3	CSTR EPNN prediction errors	49
5.4	Evolved EPNN network for CSTR	50
5.5	Two phase extraction system for PEG	55
5.6	Trained EPNN network structure for PEG system	56
5.7	Top phase PEG concentration study	58

LIST OF FIGURES
(Continued)

Figure **Page**

5.8 Tuned top phase PEG predictions 59

5.9 Phase diagram (training results) 61

5.10 EPNN training errors 62

5.11 Phase diagram (prediction results) 63

5.12 EPNN prediction errors 64

5.13 Evolved EPNN structure for biodegradation 71

5.14 Comparison between experimental data and EPNN prediction of concentration profiles for 2-chlorophenol and biomass in experiment K-13 . . 73

5.15 Comparison between experimental data and the original Andrews model prediction of concentration profiles for 2-chlorophenol and biomass in experiment K-13 74

CHAPTER 1

INTRODUCTION

Due to the technological advancements in the chemical industry, improvements from process modelling techniques are expected to lead to better process performance. The increasing emphasis on product quality, economic process performance and environmental issues in the chemical and process industries is placing significant demands on existing operational procedures (Stephanopoulos, 1990; Willis et al., 1991). Enhanced process performance generally requires more process knowledge, with mathematical models being the most common means of representing this kind of knowledge (Ungar et al., 1996; Tsoukalas and Uhrig, 1997; Jarke and Marquardt, 1996).

While it may be possible to develop a model using detailed knowledge of the physics and chemistry of a system, there are a number of drawbacks to this approach. Most chemical engineering processes are nonlinear and complex with conventional modelling and simulation techniques relying often on certain simplifying transport, kinetic and thermodynamic assumptions (Pham, 1998; Zhang et al., 1998a; McKay et al., 1997; Klein and Rivera, 2000; Cao et al., 1999). Therefore, it may take a considerable amount of time and efforts to develop a working model. Such models are often costly to develop and may be subject to inaccuracies or even uncertainties in some thermodynamic models, such as the development of a biodegradation model for environmental applications (Mandal, 1998). Furthermore, in some cases, the chemical process is so complex that there even does not exist any thermodynamic process model that can fully clarify the process, such as the modelling of some organic

multicomponent adsorption on activated carbon (Carsky and Do, 1999). However, if an accurate process model is available, then many of the benefits of improved process operability would be achievable. The current trend in process industries is to use data based modelling techniques to develop accurate, cost-effective input-output process descriptions for processes where traditional process modeling techniques perform poorly. The popular techniques may be divided into two categories. The first is based on the use of various statistical techniques and regression analysis, while the second involves the use of modern artificial intelligence, especially artificial neural networks or evolutionary computing.

The most well known and simplest way to apply statistical techniques assumes that any relationships between input and output variables are linear and that the data are normally distributed. Unfortunately, industrial systems are normally highly non-linear and the data obtained from such processes generally do not conform to normal distributions. Nevertheless, numerous methods can be used to implement a systematic data analysis methodology and can help to establish the basic characteristics of the process. However, it should be noted that a certain degree of expertise is often required in applying and interpreting such statistically based results. For example, statistical regression of vapor liquid phase equilibria requires extensive knowledge of different types of equations of state and their related modeling parameters (Zhang et al., 1998b; Blas and Vega, 1998). This is one of the reasons why researchers have so rapidly developed and improved the use of artificial intelligence techniques.

Compared to traditional statistical approaches, modern artificial intelligence methods are developed to model nonlinear, steady-state or dynamic systems with a minimum requirement of the knowledge of the system mechanism and the corresponding parameters. Among various modern artificial intelligence techniques developed for process engineering and chemistry, artificial neural networks and evolutionary computing are the most commonly used. There is an increasing amount of applications developed in the literature focusing on the applications of artificial neural networks and evolutionary computing in industrial processes modeling.

It has been shown mathematically that a neural network is capable of learning any continuous non-linear input-output mapping (Hornik et al., 1989). Indeed, applications within the chemical and process industries indicate that neural networks can adequately represent process system behaviors (Pham, 1998; Zhang et al., 1998a; McKay et al., 1997; Chiou and Wang, 1998; Roubos et al., 1999; Simutis and Lubbert, 1997). While neural networks can provide an extremely effective black-box modeling tool for both steady-state and dynamic systems, the technique has some inherent disadvantages and limitations. The first is the network structural determination, which generally involves heuristics or time-consuming iterative design techniques (Doherty et al., 1997). The second is that the modeling results are often difficult to analyze or interpret (Willis et al., 1997; McKay et al., 1997).

Since the mid 1990s, evolutionary computing has been developed and applied in the chemical and process industries. There are a number of successful examples in the literature (McKay et al., 1997; Edwards et al., 1998; Moros et al., 1996; Greeff

and Aldrich, 1998; Fraga and Matias, 1996). Compared to artificial neural network approaches, evolutionary computing modeling results provide ways to indicate the relative contribution of each input to the output through symbolic or token-based regression. However, pure evolutionary computing techniques have drawbacks in cases where feedback and saturation may appear and simple symbolic regression may fail (Zhang et al., 1998a).

Therefore, it is essential to develop a reliable and flexible modeling technique that can combine the network structure of neural networks with the symbolic regression power of evolutionary computing in the field of data based process modeling. Such a model should be at least as adaptive and flexible as existing artificial neural networks or evolutionary computing techniques in a given application.

CHAPTER 2

LITERATURE REVIEW

2.1 Overview of Traditional Methods

Prior to the development of data-driven artificial intelligence models, such as neural networks and evolutionary computing, traditional methods in chemical engineering modeling generally relied on thorough understanding and relatively complete physical and chemical knowledge of the target process. In the following discussion, the focus will be on some typical traditional chemical engineering modeling techniques and their limitations.

One example in traditional chemical process modeling is phase equilibria modeling, especially the widely used group contribution models. The basic aim of group contribution models is to utilize existing phase equilibrium data to predict the phase equilibria of systems for which no data are available. While such predictions may be used for preliminary design purposes, it must be emphasized that group contribution methods for predicting phase equilibria are of a semiquantitative nature only. Wherever good experimental data are available, these should be used rather than the group contribution predictions. Although these models have been successfully applied in many vapor-liquid phase equilibria systems, they require comprehensive knowledge of molecular surfaces and volumes of the pure components (such as the UNIQUAC model), which have to be estimated from molecular group structural contributions.

UNIFAC (Fredenslund et al., 1977) is one of the increasingly widely used *group contribution* models based on the UNIQUAC model. In spite of many successful applications, the UNIFAC model does have its limitations:

- Polymers are not included.
- Extensive knowledge of molecular properties is required: volume parameter, surface area parameter, group interaction parameter, residual activity coefficient.
- Fundamental deficiency of the *group contribution* approach : group contribution methods are developed based on sub-molecular functional groups and hence do not *know* what type of applications and what kind of molecular and process environment they are being used for. Therefore, the model validity and accuracy may be compromised by using the same group contribution model with the same set of parameters for all applications (Sandler, 1994).
- Difficult to describe highly polar systems.

Furthermore, it is not possible to combine modified group contribution equation of state tables with existing UNIFAC parameter tables, which means a whole different set of parameter tables has to be developed each time for some modified models. Though it may not be technically impossible in some cases, it may be economically prohibitive and impractical to do so. For liquid mixtures with polymers, modified UNIFAC requires a completely new parameter table, which includes the densities of

the pure solvent and pure polymer at the temperature of the mixture of interest and the structure of the solvent and polymer (Sandler, 1994).

In addition, group contribution model prediction of liquid-liquid equilibria is still highly uncertain (Sandler, 1994). In order to predict liquid-liquid phase equilibrium, it is necessary to develop a special parameter table based on liquid-liquid equilibrium data (Magnussen et al., 1981), which points out a major deficiency in the model. Any given activity coefficient model should, with the same parameters, be able to predict any type of equilibria. It is not only group contribution activity coefficient models which exhibit this weakness, molecular models do so as well.

Furthermore, many such group contribution models result in equations with more than one set of possible parameters, though only one of these sets of parameters fits the experimental data closely (Walas, 1985). For example, the widely investigated Wilson equation in infinite dilution form:

$$\Lambda_{12} = \frac{1}{\gamma_1^\infty} \exp\left(1 - \frac{1}{\gamma_2^\infty} \exp(1 - \Lambda_{21})\right) \tag{2.1}$$

In some cases, the above equation can yield three sets of parameters without any additional information except for the data used for parameter determination. Identifying the physically correct choice from multiple sets of parameters is a problem when not enough data are available beyond the few that may have been used for determination of the parameters (Walas, 1985).

All of the above problems limit the applications of group contribution in phase equilibria modeling. The demand for and rising attention to data-driven applications

in some of these non-applicable areas is one of the major driving forces of the development of modern artificial intelligence techniques in chemical process modeling. Artificial intelligence techniques provide alternative ways to partially solve the above problems, though these may not be capable of solving all of them.

Another example in traditional modeling approaches is the modeling of adsorption of binary organic vapor mixtures. Similar to vapor-liquid phase equilibria modeling, modeling and predictions of multicomponent adsorption of organic vapor on activated carbon are very desirable due to the extremely time-consuming experimental measurements of adsorption behavior (Brasquet and Cloirec, 1999). Traditional empirical and theoretical models have been developed to predict the adsorption equilibria based on single component adsorption data. For example, the Ideal Adsorption Solution (IAS) model assumes that the mixture pressure depends only on the concentration and pressure of the pure components and their isotherms. While for some binary systems, such assumptions and calculations are quite accurate, there are substantial deviations in many other systems (Carsky and Do, 1999). The actual process is quite complex for some systems and has not yet been fully clarified using the existing thermodynamic and transport models. While there are recent developments trying to overcome such limitations through time-consuming experimental efforts (Myers et al., 1992), more attention is paid to the use of data-based approaches, such as artificial neural networks, to provide accurate and feasible data-based process models (Carsky and Do, 1999).

Furthermore, in some process modeling cases, it is extremely difficult to determine the parameters of a traditional model, such as the determination of the Andrew's parameters in a biodegradation process (Mandal, 1998) through a number of trials and *guessing*. Alternative methods for producing empirical formulae for the studied process without arbitrary assumptions is highly desirable.

Therefore, it is necessary to develop data based modeling techniques in addition to existing traditional models to completely model various chemical processes.

2.2 Overview of Artificial Intelligence

Artificial intelligence(AI), in a broader sense, encompasses a number of technologies that include, but are not limited to, expert systems, neural networks, evolutionary programming, fuzzy logic systems, cellular automata and chaotic systems. They are part of the *soft computing* branch of science and engineering (Tsoukalas and Uhrig, 1997). Many of these technologies have their origins in biological or behavioral phenomena related to human or animal living systems. Hybrid intelligent systems generally involve two, three, or more of these individual AI technologies that are either used in series or integrated in a way to produce advantageous results through synergistic interactions.

As part of *soft computing*, artificial intelligence methods have the distinguishing characteristic that they provide approximate solutions to approximately formulated problems (Aminzadeh and Jamshidi, 1994). Though there are many different constituents of artificial intelligence, such as fuzzy logic, neural networks, expert systems

and evolutionary computing, all branches of AI have the same common feature of tolerating imprecision and uncertainty in order to develop more tractable and robust models of systems at a lower cost and greater economy of communication and computation. In this dissertation, the developed method is based on existing artificial neural networks and evolutionary computing methods. Therefore, the following overview will be focused on these two branches of artificial intelligence.

2.2.1 Artificial Neural Networks

It has long been known that learning in animals and humans can be achieved through observation of examples. The exact mechanism by which this learning takes place is still unknown, but science has yielded some clues. In 1909, Cajal Omatu et al. (1996) found that vertebrate brains consist of an enormous number of interconnected cells called neurons. It has since become widely accepted that these neurons are the fundamental information processing elements of brains.

Basically, a living biological neuron receives inputs from other sources, combines them in some way, performs a generally nonlinear operation on the result and finally outputs the final result. Figure 2.1 shows a sketch of a biological neuron showing the most important components.

Artificial neural networks are relatively crude computing models based on the neural structure of a living brain. Artificial neurons work in a way similar to biological neurons. The brain basically learns from experiences. The brain modeling promises a less technical way to develop machine solutions.

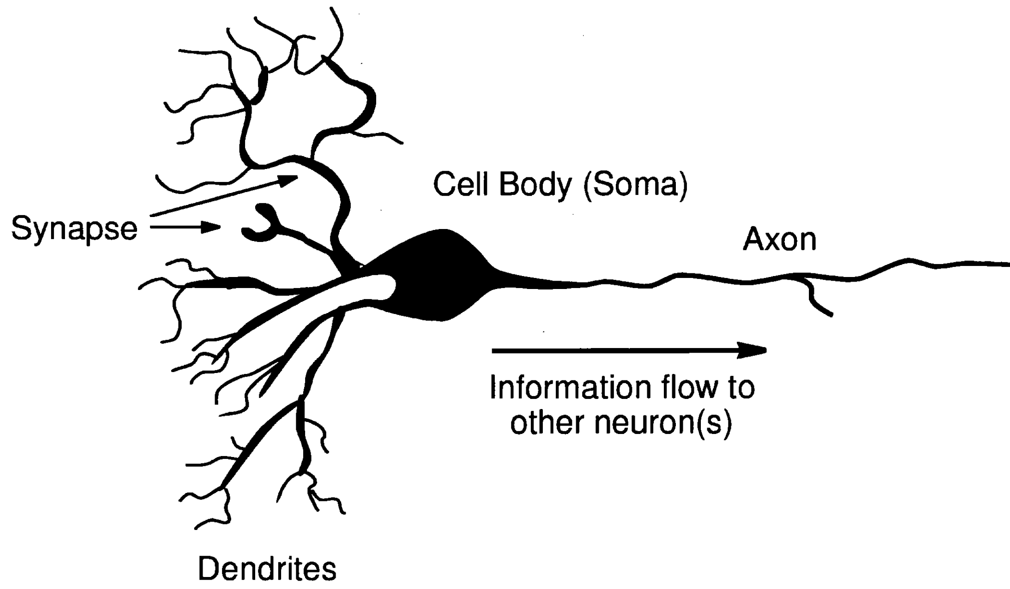

Figure 2.1: Sketch of a biological neuron

In a nutshell, an artificial neural network (ANN) can be defined as follows (Umeda and Niida, 1986): A data processing system consisting of a large number of simple, highly interconnected processing elements(artificial neurons) in an architecture inspired by – but not limited by – the structure of cerebral cortex of human brain.

ANN were originally created as an attempt to model the act of thinking by modeling neurons in a living brain. The history of neural networks can be dated back as early as the 1940s when McCulloch and Pitts introduced the first neural network computing model. In 1956, Rosenblatt's work resulted in a two-layer network, the perceptron, which was capable of learning certain classifications by adjusting connection weights. Although the perceptron was successful in classifying certain patterns, it had a number of limitations. The perceptron was not able to solve the

classic XOR (exclusive or) problem. Also the limit of computing facilities largely hindered the development and research on neural networks. Such limitations led to the decline of the field of neural networks. However, the perceptron had laid foundations for later work in neural computing.

During the mid 1970s to 1980s, researchers showed renewed interest in neural networks. The first practical learning method for neural network, backpropagation, was developed at this time (Werbos, 1974; Rumelhart et al., 1986). Prior to the development of backpropagation, attempts to use perceptrons with more than one layer of weights were frustrated by the "weight assignment problem". This problem plagued the neural network field for over two decades. Since the 1990s, backpropagation neural networks have been widely used in many fields, such as nonparametric modeling (Kan and Lee, 1996), dynamic system forecasting (Zhang et al., 1998a), system control (Ungar et al., 1996) and pattern recognition (Rivera and Klein, 1997).

Figure 2.2 shows a typical structure of an artificial neuron. In Figure 2.2, there are four direct inputs and one feedback. The processing element acquires these inputs and then processes them in a certain way and then generates the output. In the artificial neuron example in Figure 2.2, the processing element simply sums all of the acquired inputs and then uses a transfer function to produce the output. The number of inputs, number of feedbacks and type of transfer function can be different under difference application circumstances. In early works , the transfer function was a simple threshold (indicator) function rather than the sigmoidal functions commonly used today (McCulloch and Pitts, 1943). Threshold activations were found to have

severe limits and thus sigmoidal activation became widely used instead (Anderson, 1982). The transfer function is usually nonlinear. The most commonly used transfer function in ANN is the sigmoidal function, which can be described by Equation 2.2.

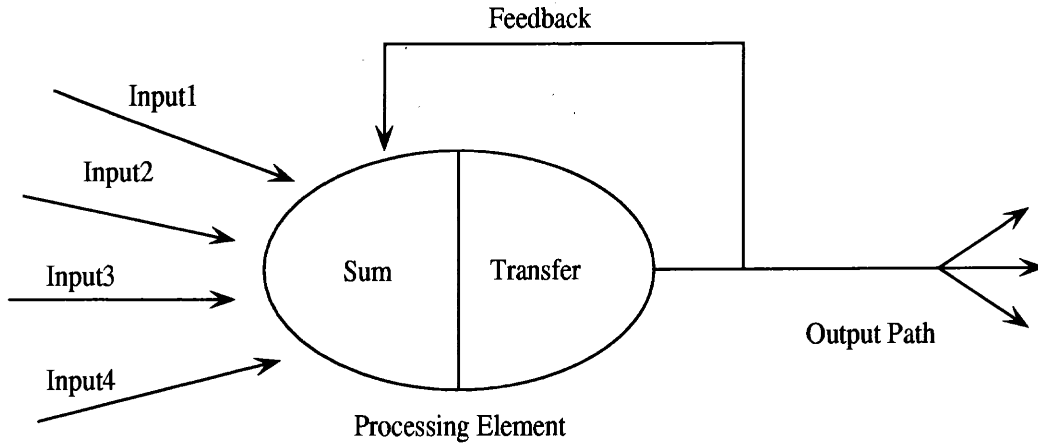

Figure 2.2: A typical structure of artificial neuron

$$\sigma(Sum) = \frac{1}{1 + \exp(-Sum)} \tag{2.2}$$

Where Sum is:

$$Sum = \sum_i (Input_i)$$

Current ANNs are just a simple clustering of primitive artificial neurons. This clustering occurs by creating layers which are then connected to one another. How these layers connect is one of the major factors that distinguishes different types of ANNs.

In most networks, each neuron in a hidden layer receives the signals from all of the neurons in a layer before it, typically an input layer. After a neuron performs its transfer function, it passes its output to some of the neurons after it, providing a

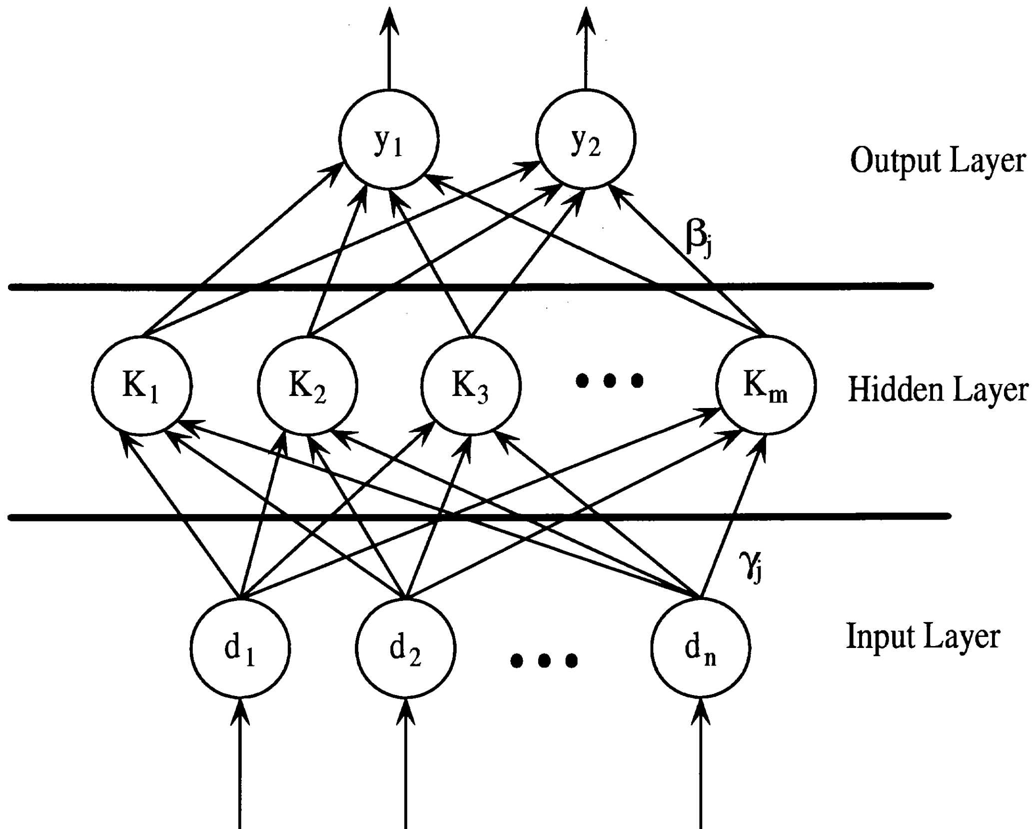

Figure 2.3: A sample feedforward neural network

feedforward path to the output, as shown in Figure 2.3. This type of networks is a feedforward neural network.

Another type of connection is a feedback link in recurrent neural networks. For feedback links, the output of one layer routes back to a neuron in the previous layer. An example of a recurrent neural network is shown in Figure 2.4. While feedforward neural networks are easier to implement, they have no "memory" since the output at any instant is dependent solely on the inputs and the link weights at that instant, as stated in the literature (Tsoukalas and Uhrig, 1997).

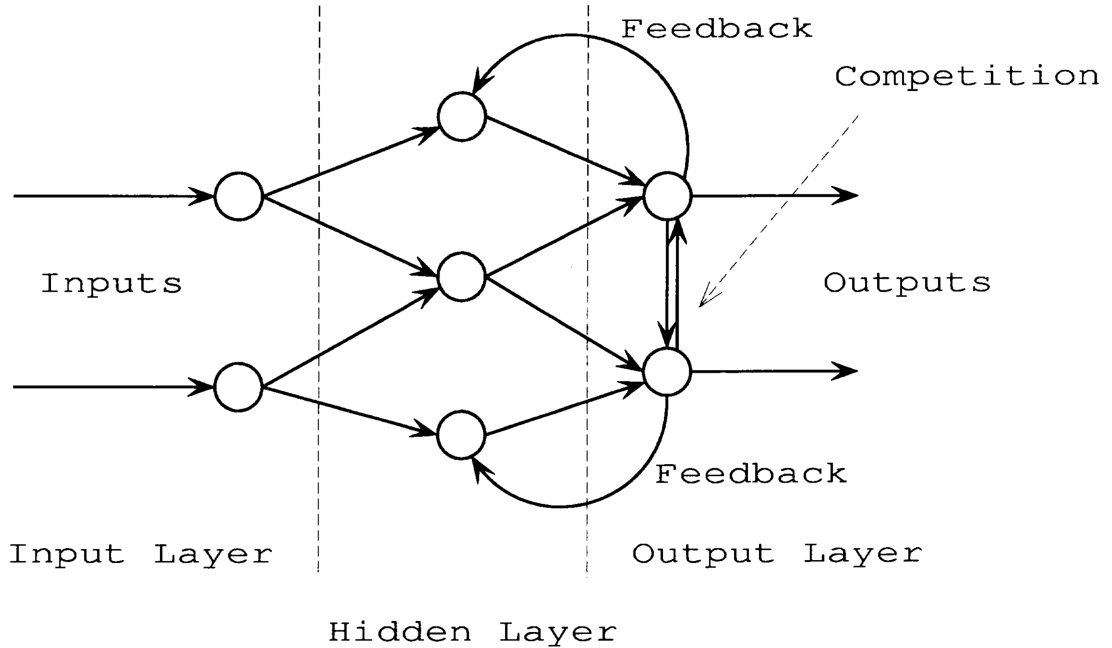

Figure 2.4: A sample recurrent neural network

Much of the recent work on neural networks stems from a number of papers, including Funa-hashi (1989) and Hornik et al. (1989) that showed that neural networks are a way to approximate an arbitrary function closely as the number of hidden nodes gets large. Neural networks work well with continuous functions. Some such continuous functions are very complex and are extremely difficult or even impossible to handle by traditional modeling techniques.

2.2.2 Evolutionary Computing

Evolutionary computing was originally initiated and developed as a searching concept for solving difficult optimization problems. Though the earliest field in evolutionary computing, genetic algorithms, was initiated in the early 1970s (Holland, 1975),

their applications in real world practical problems was developed almost two decades later. This was largely due to the lack of available powerful computer platforms at that time, but was also due to some methodological shortcomings of those early approaches (Fogel, 1995a). Goldberg presented the first systematic research collection on genetic algorithms in real world applications (Goldberg, 1989). In 1992, John Koza used genetic algorithms to evolve programs to perform certain tasks. He called his method *genetic programming*(GP). The origin of genetic programming significantly advanced the development of genetic algorithms, allowing evolutionary computing methods to not only search and optimize numerical systems but also to generate symbolic formulae and schematic programs to automate and evolve the computing process itself. A more detailed and complete overviews of the history of evolutionary computing can be found elsewhere (Fogel, 1998; Bäck et al., 1997).

Evolutionary computing is developed as a set of concepts for problem solving rather than a collection of related and ready-to-use algorithms. The majority of current implementations descend from three strongly related but independently developed categories: *evolutionary algorithms*, *evolutionary programming* and *evolutionary strategies* (Bäck et al., 1997). The most commonly used methods in engineering are evolutionary algorithms, which consist of genetic algorithms and genetic programming. This work will focus on the discussion on evolutionary algorithms, especially genetic programming.

Typically, evolutionary algorithms contain a randomly or deterministically generated population of individuals. Each individual represents a potential solution for

a given problem. During evolution, each individual receives a measure of its fitness for the given problem. Based on the fitness, the fitter individuals are selected and put into crossover and mutate to generate the next generation in the population. In some evolutionary computing cases, older generations are discarded and replaced by newer generations. This evolutionary process continues from generation to generation until the desired fitness or satisfactory solutions have been reached by the fittest individuals. Figure 2.5 shows a typical evolutionary computing algorithm.

```
produce an initial population of individuals
evaluate the fitness of all individuals
  while   termination condition not met    do {
      select fitter individuals for reproduction
      recombine individuals
      mutate some individuals
      evaluate the fitness of the new individuals
      generate a new population by inserting some
        new good individuals and by discarding some old
        bad individuals
  }
  end while
```

Figure 2.5: A typical evolutionary algorithm

One major advantage of evolutionary computing is that it is conceptually simple. The desired solutions are reached by repeated crossovers and mutations. During crossovers and mutations, partially better solutions are generated.

In genetic programming, each evolving individual is a computing formula or *program*, while in genetic algorithms each individual is a fixed length string of numbers, or *vector*. Generally speaking, the string representation in genetic algorithms encodes each target process parameter as a variable in a multiple dimensional vector.

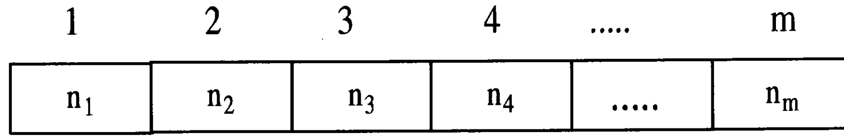

Figure 2.6: An example of string representation of numbers in genetic algorithms

An example of string representation of numbers is shown in Figure 2.6. The corresponding vector of the genetic algorithm individual is:

$$(n_1, n_3, n_3, n_4, \ldots, n_m)$$

where n_i is a real value for a target variable. Mutation and crossover in genetic algorithms can involve one or many points in the string representation. Figure 2.7 shows an example of single point mutation in genetic algorithms and Figure 2.8 shows a single point crossover in genetic algorithms.

As a contrast, in genetic programming, each individual represents a computing formula in various tree forms (Banzhaf et al., 1998; Koza, 1992, 1994). An example of a tree representation of algebraic formulas is shown in Figure 2.9. The corresponding algebraic formula for the tree representation in Figure 2.9 is Equation 2.3.

$$\frac{x_2}{3.1} - \ln(x_1) + \exp(x_1 + x_2) \times (5 + x_1) \tag{2.3}$$

The variable x_1 and x_2 can be input variables of the modeling system. The corresponding depth of the formula tree for Equation 2.3 is 4.

Figure 2.7: An example of single point mutatation in genetic algorithms

Figure 2.8: An example of single point crossover in genetic algorithms

During crossover in genetic programming, two individuals are selected. A random crossover point is selected in the tree of each individual. Then the subtrees below the crossover point in each individual are exchanged. Therefore, two new individuals are generated. Mutation is implemented by randomly removing a subtree in an individual and replacing it with a newly randomly generated subtree. Therefore, the evolving techniques and results are different from the methods used in genetic algorithms.

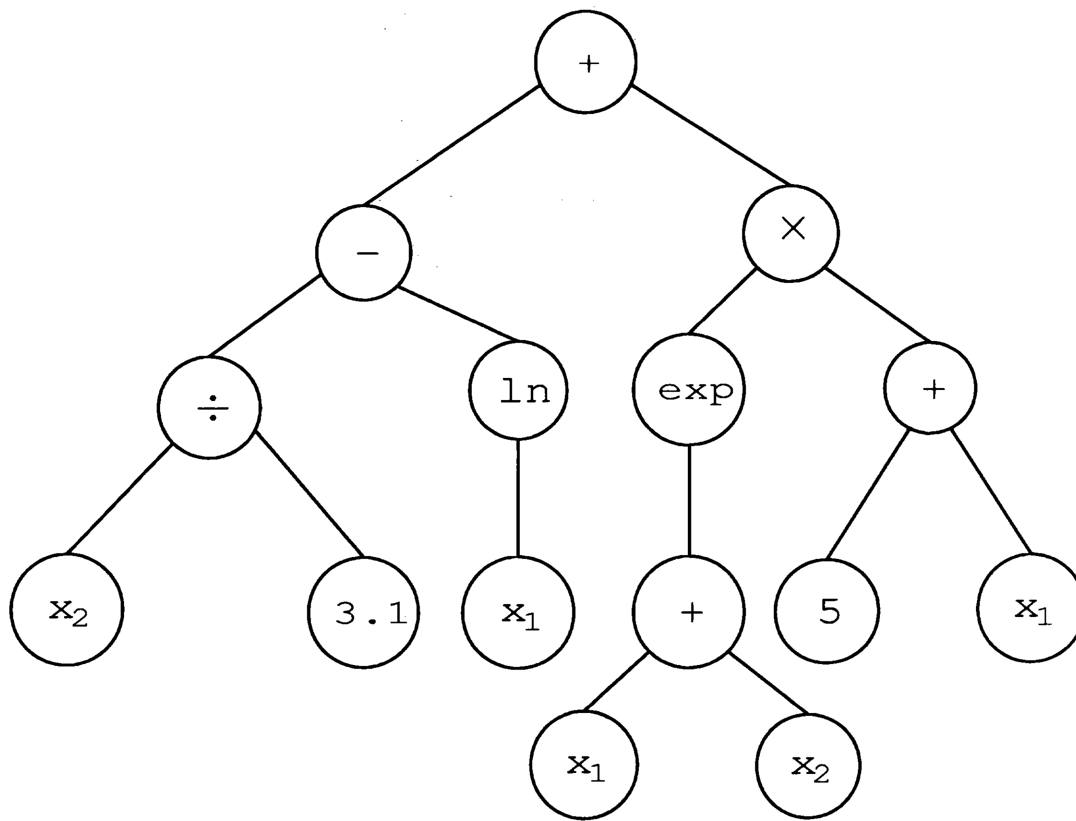

Figure 2.9: An example of a tree representation of computing formula

More thorough discussions about the difference between genetic algorithms and genetic programming can be found in the literature (Fogel, 1995a; Koza, 1992; McKay et al., 1997; Greeff and Aldrich, 1998). Detailed introductions and practical algorithms in genetic programming can also be found elsewhere (Banzhaf et al., 1998; Koza, 1992, 1994).

2.2.3 Current Trends in Artificial Intelligence

Since the application-oriented development of neural networks and evolutionary computing, both fields have been advancing into more adaptable hybrid techniques. A

number of hybrid methods based on ANN and evolutionary computing have been proposed and applied in several engineering fields, such as evolving neural networks with evolutionary computing techniques (Yao and Liu, 1997; Fogel et al., 1990), or using a neural network structure with genetic algorithms to adapt its parameters (Gao et al., 1999; Ghosh et al., 2000; Zhao et al., 2000).

Hybrid approaches have proven to be more advantageous in many fields over their pure counterpart techniques. It is believed artificial intelligence will include more new hybrid methods in the future.

2.3 Applications in Chemical Engineering

During the past decade, a number of artificial neural networks and evolutionary computing techniques have been developed and applied in chemical engineering processes, such as kinetics modelling (Edwards et al., 1998; Galvan et al., 1996; Moros et al., 1996; Cao et al., 1999), dynamic process forecasting (Pham, 1998; Zhang et al., 1998a), phase equilibrium modeling (Kan and Lee, 1996; Sharma et al., 1999), steady-state chemical process modeling (McKay et al., 1997), bioprocess modeling and optimization (Chiou and Wang, 1998; Roubos et al., 1999; Simutis and Lubbert, 1997) and bioseparation modeling and optimization (Klein and Rivera, 2000). There are also applications in adsorption, bio-degradation, and assisting chemical structure determination. A thorough overview of applications of artificial intelligence in engineering can be found elsewhere (Fogel, 1995a).

In early application development of artificial intelligence in chemical process modeling, the approaches of artificial neural networks and genetic programming were developed along separate paths. Neural networks were generally applied to system identification and modeling and long-term prediction and control (Zhang et al., 1998a; Kan and Lee, 1996; Doherty et al., 1997), while evolutionary computing was mostly applied to steady state modeling (McKay et al., 1997; Greeff and Aldrich, 1998). In the past few years, there have been some applications of hybrid methods based on both ANNs and genetic algorithms in chemical and biochemical process modeling and optimization (Ghosh et al., 2000; Zhao et al., 2000; Gao et al., 1999). However, none of them has combined the symbolic regression feature of genetic programming with neural network structures, though there exist some hybrid methods in non chemical engineering applications (Yao and Liu, 1997; Chen and Ni, 1997). More recent efforts that are trying to combine artificial neural networks with genetic programming in non-chemical engineering fields can be found in the literature over the past two or three years (Segovia and Isas, 1998; Kiguchi et al., 1999; Yeun and Lee, 1999). However, in those efforts, genetic programming is still used only as tuning methods to tune up the training algorithms of traditional neural networks.

Therefore, motivated by the potential synergy resulting from both neural networks and genetic programming, a novel approach has been developed and is presented in this dissertation. The proposed model combines the feedback structure of recurrent artificial neural networks and the symbolic evolving methods of genetic programming. The model can produce empirical symbolic formulae for dynamic

systems, a feature that is not existent in either ANN or genetic programming. The produced empirical symbolic formulae can be used for theoretical analysis and process optimization. Furthermore, the proposed method can model the bioprocesses which can be hardly modeled accurately without arbitrary assumptions by traditional models.

CHAPTER 3

OBJECTIVES

Upon research on the limitations of traditional process modeling techniques and existing artificial neural network and evolutionary computation methods, the primary objective of the dissertation was to develop an alternative method that is capable of modeling chemical processes using a data-based approach.

In addition to producing more accurate and efficient results, the proposed model is expected to overcome some of the limitations of existing methods, such as the "blackbox" nature of neural networks and the limits of genetic programming for modeling complex processes. Furthermore, the proposed model is also developed to be used as a feasible method to provide empirical models for some of the processes for which traditional methods can hardly yield satisfactory results or for processes for which the determination of traditional modeling parameters is difficult.

Specifically, the following are included to complete the objectives:

- Complete descriptions of the composition of proposed modeling system and related algorithms (Chapter 4).
- Full discussion of modeling parameters in the proposed model and their application considerations for chemical processes (Chapter 4 and Chapter 5) .
- Some typical applications in chemical process modeling of the proposed model. Comparisons with either existing traditional neural network models or traditional thermodynamic models (Chapter 5).

- Demonstration of the advantages of using the proposed model compared to existing techniques (Chapter 5 and Chapter 6).

- Study and give the limitations of the proposed model and suggest possible improvements based on recently published results (Chapter 7).

CHAPTER 4

EPNN MODELING

4.1 Introduction

As discussed in previous chapters, especially Section 2.3, it is highly desirable to develop a new hybrid method based on both neural networks and genetic programming to model complex chemical processes.

Introduced below is such a novel approach motivated by the potential synergy resulting from both neural networks and genetic programming. The proposed model combines the feedback structure of recurrent neural networks and the symbolic evolving methods of genetic programming. This model can produce sets of empirical symbolic formulae for both dynamic and steady-state systems through evolutionary training. The produced empirical symbolic formulae can be used for theoretical analysis and process optimization.

The presented model, Evolutionary Polymorphic Neural Network, or EPNN, is mainly based on recent artificial neural network and genetic programming (GP) developments. However, it must be stipulated that GP is very different from genetic algorithms(GAs). Also, while the proposed model has features from GP, it is not a replication of GP. In the proposed model, genetic algorithms are not employed. Discussions on the differences between GA and GP can be found in Section 2.2.2.

The EPNN modeling system mainly inherits two features of GP. One feature is the evolution of randomly or deterministically generated population of individuals. The other feature is the tree representation of symbolic formulae in each individual.

Descriptions of how to represent a symbolic formula in a tree form was discussed in Section 2.2.2 and Figure 2.9 shows an example of such a representation. The corresponding depth of the formula tree in Figure 2.9 is 4. Discussions of formula trees and depth measurements can also be found in the literature (Aho et al., 1987; Greeff and Aldrich, 1998; McKay et al., 1997).

Before demonstrating the applications of EPNN in chemical process modeling, it is necessary to give detailed descriptions of modeling system structures, algorithms and their implementations.

4.2 General Model Description

4.2.1 System Structure

A typical composition of a complete EPNN modeling system is shown in Figure 4.1. Working vertically downward in Figure 4.1, a complete EPNN system consists of necessary methods such as evolutionary algorithms, tuning methods and individual survivor strategies. The EPNN system also includes the target data structure which is a population of EPNN individuals. This population is further divided into two sub-populations: *gene pool* and *mutation pool.* The *gene pool* holds candidate solutions while the *mutation pool* acts as an auxiliary during evolution. Each pool is a collection of EPNN individuals and each individual is a highly interconnected network of symbolic formulae. An EPNN network is defined with four sets of symbols: function set, input variables set, output variables set and intermediate nodes (neurons). Intermediate or output nodes are associated with symbolic formulae. Figure 4.2 shows an example of an EPNN network with two intermediate nodes and one

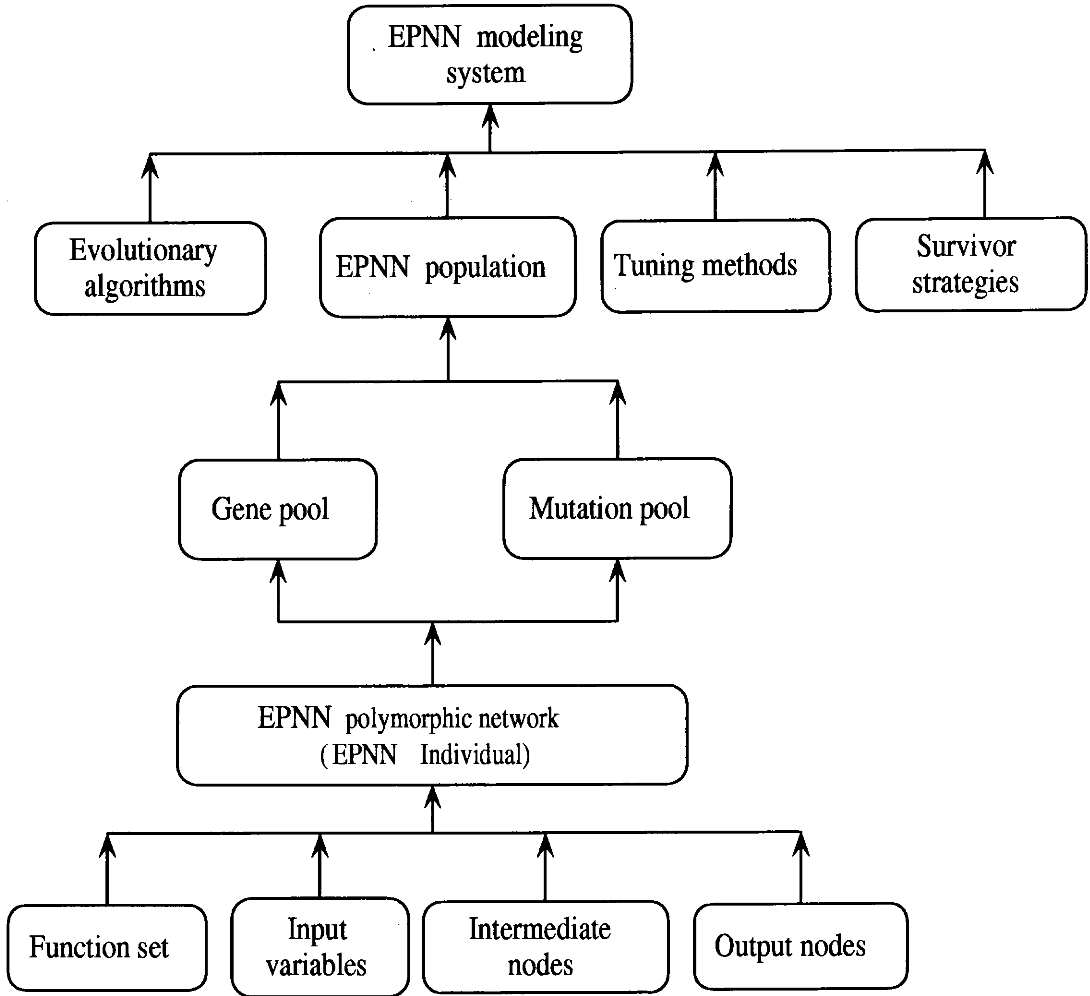

Figure 4.1: A typical composition of EPNN modeling system

output node. The corresponding functional form of the network is:

$$N_0 = f_0(N_0, N_1, N_2, Input)$$

$$N_1 = f_1(N_2, Input)$$

$$N_2 = f_2(N_1, N_2, Input) \tag{4.1}$$

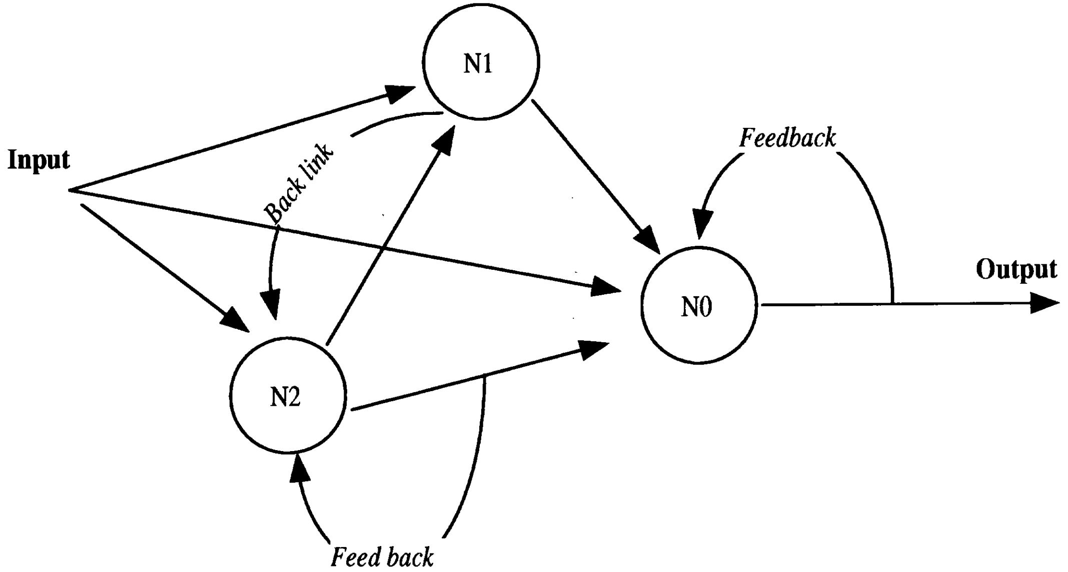

Figure 4.2: A sample EPNN individual network

One of many possible symbolic formulae that the above functional form may stand for is:

$$\begin{aligned} N_0 &= \frac{X + N_0}{N_1 + N_2} \\ N_1 &= \frac{N_2}{X + 0.8562} \times (0.2345) \\ N_2 &= N_2 \times X + 0.1745 \times \ln(N_1) \end{aligned} \quad (4.2)$$

The function set here is : $\{+,-,\times,\div$ and ln $\}$. The input variable is X and the output variable is N_0. The intermediate nodes are N_1 and N_2. Each node (N_0,N_1 or N_2) is a tree representation of a symbolic formula as demonstrated in Figure 4.3. The recursive functions (N_0 and N_2) represent feedback linkage. The purpose of the feedback linkage is to provide state variables and to model potential delay responses in the target system. Each individual in an EPNN population encodes a potential

solution for the given problem by representing the potential solution with a system of empirical equations.

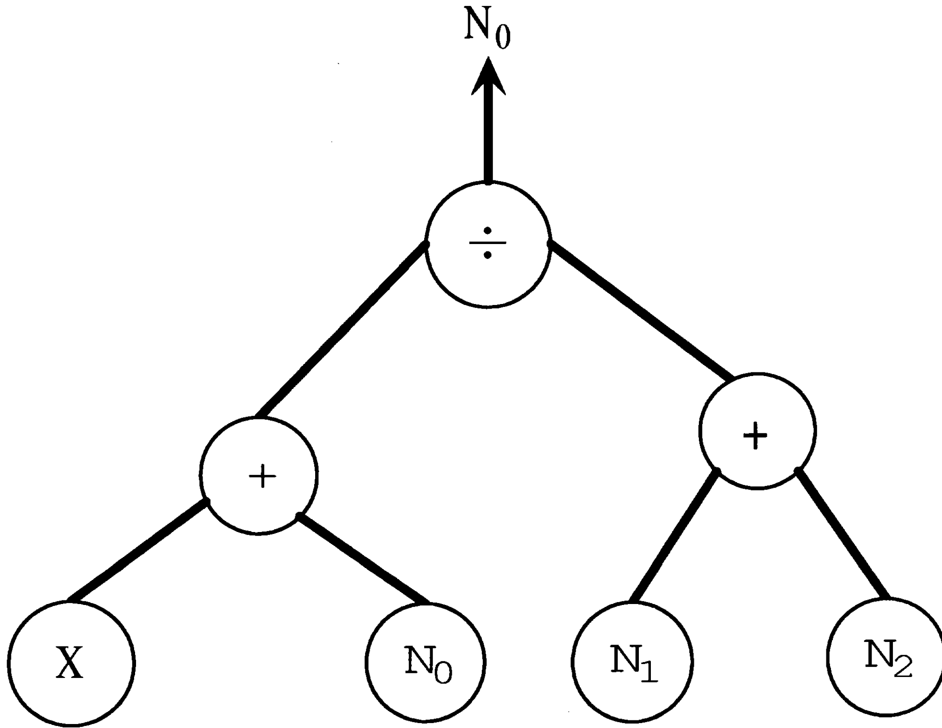

Figure 4.3: Tree representation for the formula of node N_0 in Equation 4.2

In summary, an EPNN system consists of a randomly or deterministically generated population of individuals. Each individual in the EPNN system is a highly interconnected network of symbolic formulae. Each such EPNN network is defined with four sets of symbols: function set, input variables set, output variables set and intermediate nodes (neurons). Every distinct symbolic formula is associated with one intermediate or output node.

4.2.2 Modeling Algorithms

Like existing GP and artificial neural networks, EPNN must be trained prior to data prediction. A typical training algorithm for an EPNN system is shown in Figure 4.4. At the beginning of the training step, every individual network in the population is randomly generated. In each individual, the nodes are initialized with random constants but no linkage to each other. Upon detailed examination, one can see that the values of initial random constants do not affect the final training result in a given problem. This fact is demonstrated in the investigated cases discussed in Chapter 5.

There are three levels of mutation in the training method of an EPNN system: elementary, intermediate and macro mutations. Elementary mutation is the lowest level of mutation. It deals with single node mutation, such as add, change, swap and deletion of a single constant or a single linkage. Figure 4.5 shows some typical operations of elementary mutation in tree form. Intermediate mutation deals with swap, replacement and deletion of a subtree of an EPNN formula, while macro mutation deals with swap and replacement of whole intermediate nodes. Replacement means to replace the old nodes or constants by newly randomly generated ones.

In addition to mutations, there are two levels of crossover. One is the swap of subtree during intermediate mutation. Such crossover occurs during intermediate mutations. An example of crossover operation between two tree represented formulae is shown in Figure 4.6. The corresponding system of equations (Equations 4.3) are

changed to Equations 4.4.

$$\begin{aligned} y_1 &= a \times b + 3 \\ y_2 &= \frac{a}{2} + b \end{aligned} \tag{4.3}$$

$$\begin{aligned} y_1 &= a \times b + \frac{a}{2} \\ y_2 &= 3 + b \end{aligned} \tag{4.4}$$

The other level of crossover is an individual level crossover, which deals with the exchange of one whole formula tree between different individual networks. An example of an individual level of crossover is shown in Figure 4.7. It can be seen from Figure 4.7 that two intermediate nodes were exchanged during the operation.

The three levels of mutation mentioned result in the polymorphic nature of an EPNN network. All of the above mutations have an equal chance to occur. In addition, the individual level crossover happens for every individual in the mutation pool before mutation. Two individuals are randomly selected during each such crossover. After crossover and mutation, fitter individuals in the mutation pool are then inserted into the gene pool according to their fitness (selective insertion in Figure 4.4). Then the whole population in the gene pool will be sorted again. The mutation pool here acts as an auxiliary population and will be completely regenerated each time following selection from the gene pool.

The criterion of individual survivor strategy is defined by a comprehensive fitness function. This function consists of two components: a specific life span for each individual and a combined minimum error (CME) function. The life span is

used as a penalty function for each individual and is a variable during the training step (Equation 4.5). It is a function of diversity of the whole population. The more diverse the population is, the longer life span each individual will have. The life span is defined as:

$$L_{i,j} = K \times (G_j - H_i)/D_j \tag{4.5}$$

where K is a proportionalal constant, H_i is the generation number when the ith individual was created, G_j is the jth generation number and D_j is the diversity value of jth population. Diversity is evaluated by the standard deviation of the CME function of the gene pool. The CME function is defined as:

$$CME = (R_1 \times R_2 \times R_3 \times \ldots \times R_n)^{(1/n)} \tag{4.6}$$

where R_i is defined by:

$$R_i = \sqrt{\frac{\sum_{j=1}^{m} (u_j - v_j)^2}{m}} \qquad i = 1 \ldots n \tag{4.7}$$

and m is the number of data points in each case, u represents experimental data and v represents model predictions. The quantity n is the number of outputs in the presented case. In some cases, to simplify the evolution process, the fitness function is set to include CME only, such as the modeling of a biodegradation process discussed in Chapter 5.

Once having reached desired fitness, the training algorithm is stopped. The fittest individual network is then selected as the final solution for the system. The next step is testing the prediction from the fittest individual network for forecasting or predicting the target system.

It must be pointed out that the number of random constants (such as the constant 0.2345 in Equation 4.2) in an EPNN network formula is changeable during training due to the polymorphic nature of EPNN networks. Therefore, unlike a traditional neural network, EPNN can adapt to different systems without over fitting the data.

4.2.3 Major Modeling Parameters

Prior to the application of an EPNN system, modeling parameters must be determined. Parameters that require the most attention are: function set, depth of tree and its limitations, and number of total nodes. Number of input variables and number of output variables are predetermined for each specific case. In addition, all the input and output data should be normalized to a reasonable range. In some cases, normalization can be simple rescaling and in other cases it can be a more complex function.

In order to reproduce the model in parts or in total, two special considerations need to be addressed. The first one is the maximum depth of a formula tree. Because each formula in an EPNN network has a tree structure representation, the maximum depth of a tree needs to be restricted during evolution to avoid excessively large and complicated unstable functions. On the other hand, if the restricted maximum depth of tree representation is too small, the EPNN cannot converge or will converge very slowly. In the studied cases presented in the next chapter, the maximum depth of any formula tree is limited to 100. The maximum is approximated by three pre-screening

trials according to their fitness convergence speed and the efficiency of convergence to a global maximum. The second consideration is the appropriate design of the function set. In the presented cases, the function set is {+,−,×, ÷, *ln* and *exp* }. But, in other cases, it could include sine, cosine or other periodic functions as needed for periodic or periodic-like experimental data. Therefore, before starting the training of an EPNN system, the set of functions must be properly designed.

4.2.4 Comparison with Other A.I. Techniques

In traditional neural network modeling, the linkage structure must be determined prior to training and application of the model. This can be accomplished either manually or by applying a number of trials which demands a long time and substantial effort and experience (Doherty et al., 1997; Nelson and Illingsworth, 1990; Bailey and Thompson, 1990). However, EPNN can form linkage structures dynamically during the training process through evolution, which reduces the time and effort in structural determination. During evolutionary computing (i.e. genetic programming), only one level of mutation is employed (Fogel, 1995a; Koza, 1992), while EPNN has three levels of mutations. Increasing the number of mutation levels results in more adaptable and efficient structures.

In more recent efforts, GAs were applied together with ANNs to reduce the inefficiency of ANN structure determination (Ghosh et al., 2000; Zhao et al., 2000; Gao et al., 1999). However, the "blackbox" nature of ANN was not eliminated. The model proposed here differs from the two separate modeling methods presented

by Ghosh, which use ANN as a forward modeling approach and GA as a reverse modeling method. EPNN is also different from the methods which use GA only as a way to refine ANN parameters (Zhao et al., 2000; Gao et al., 1999). The proposed EPNN network differs from these approaches fundamentally, because it is a natural combination of genetic programming and ANN. Therefore, unlike most ANNs or ANNs with GAs, EPNN does not work as a "blackbox" and can produce formulae during modeling.

4.3 Model Implementation

The general EPNN modeling system is implemented in C^{++} code by using Microsoft Visual C^{++} (TM) Version 6.0 (SP3) under Win32 platforms. The hierarchy coding scheme is illustrated in Figure 4.8. The whole system consists of the following structural components:

- Random number generator
- Main EPNN driver
- Initialization module
- Parameter module and data pool module
- TreeNode, EPNN individual and EPNN pools
- Individual evolutionary strategies (elementary and intermediate mutations as well as intermediate crossovers)

- Pool evolutionary strategies (macro mutations and macro crossovers)
- Fitness and prediction calculation module
- Data file I/O module

The system starts from the initialization module. The module includes a generation of specific size of gene pool and mutation pool of EPNN individuals. In order to guarantee uniform network structure distribution of the individuals, all individual networks are initialized with no linkage to internal nodes but possible linkage to inputs. During initialization, each node in an individual network has only the simplest form of random symbolic formula.

The random number generator module is a critical part in the EPNN evolutionary strategies. The module is implemented partially based on the random number generating techniques introduced from the literature — *Numerical Recipes in C* (Press et al., 1993) with an adjustment to allow generating time-dependent random numbers. The detailed source code for the random number generator is listed in Appendix A.

The hierarchy scheme from elementary data to the EPNN model is implemented in a standard C^{++} class composition chain. An EPNN model consists of a certain number of EPNN individuals. Each individual consists of a certain number of tree nodes, which are implemented using tree structures. Each tree node contains elementary data, which can be a generated random constant, or a link to another node. The detailed scheme of the hierarchy system can be found in Appendix A.

Both the gene pool and mutation pool are implemented via standard C^{++} vector containers. Each container holds a certain number of individual EPNN networks. Sorting and selection algorithms are developed based on the standard C^{++} sorting and selection methods. Source code for sorting and selection in both pools are listed in Appendix A. Similar to the class hierarchy, the mutations and crossovers are also implemented in a hierarchy scheme, starting from elementary to macro mutations.

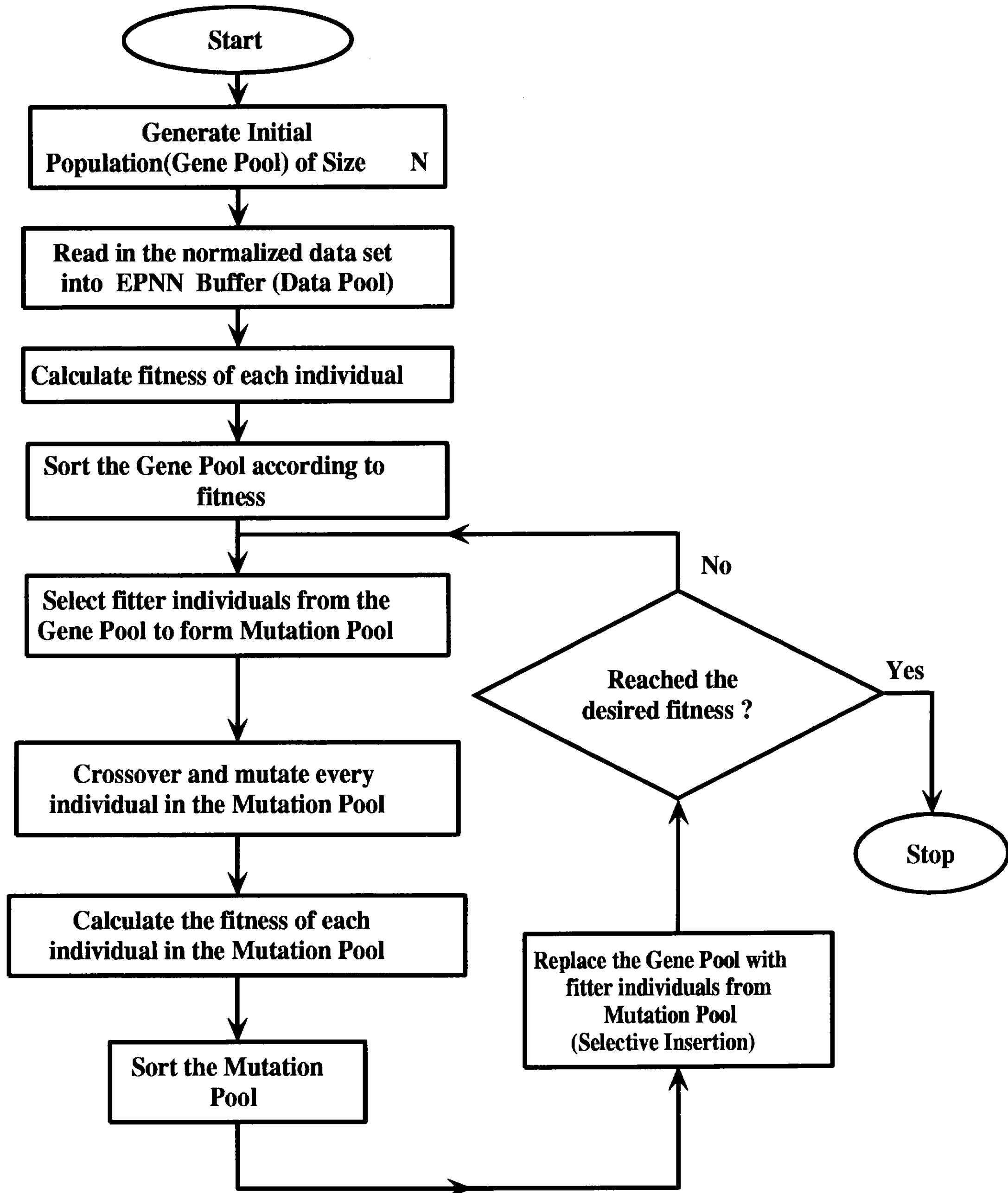

Figure 4.4: A typical EPNN network training algorithm

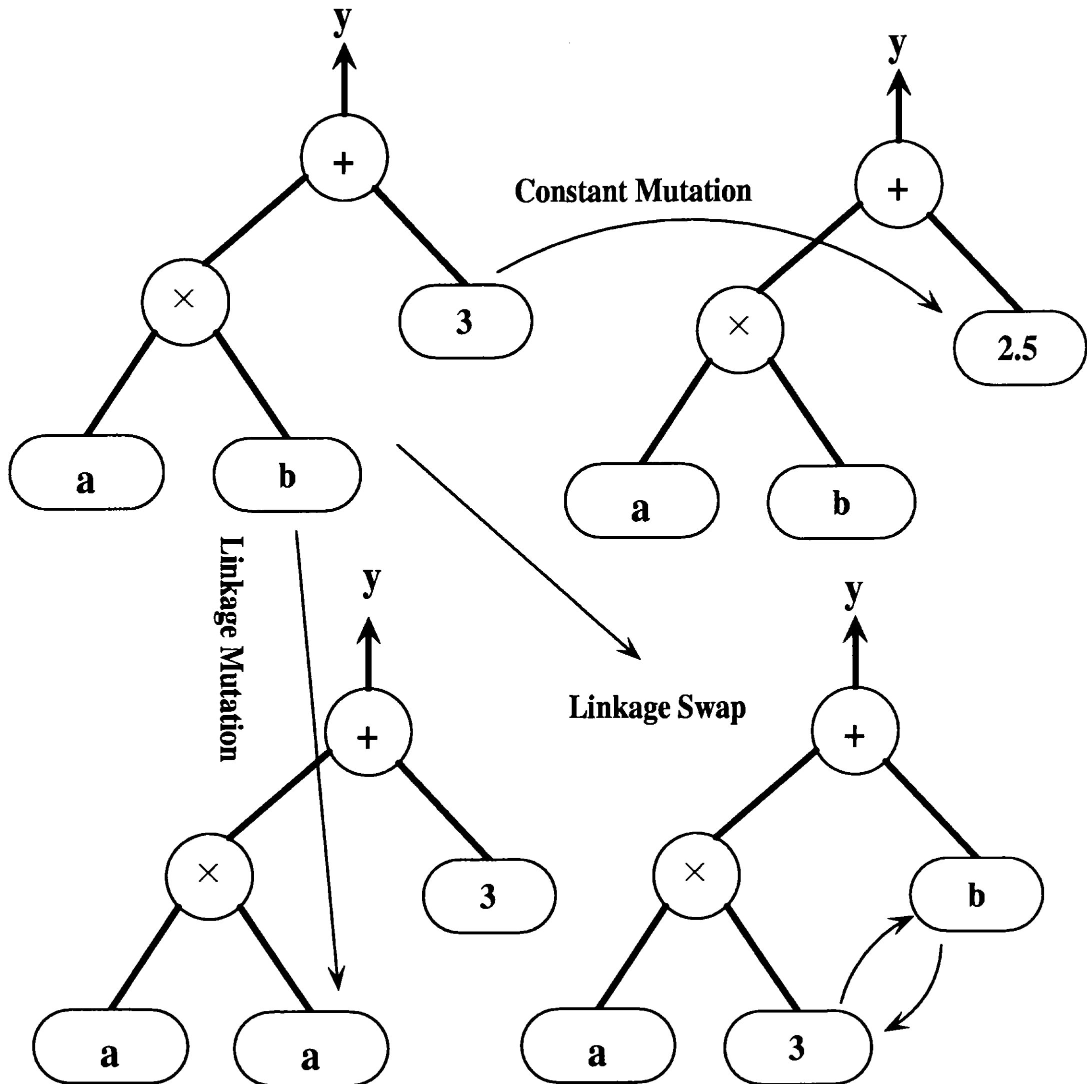

Figure 4.5: Some typical operations during elementary mutation

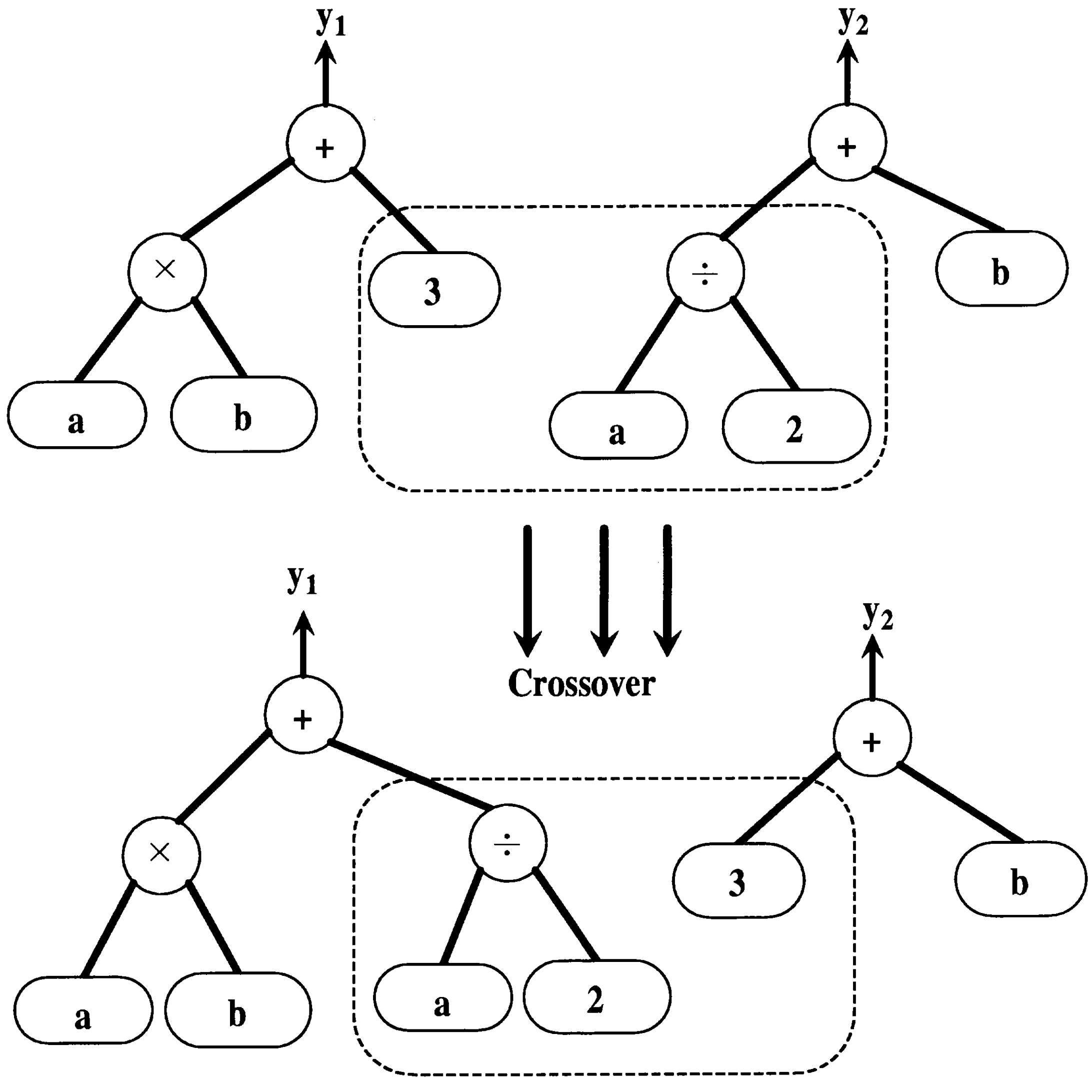

Figure 4.6: A typical crossover operation in tree representation

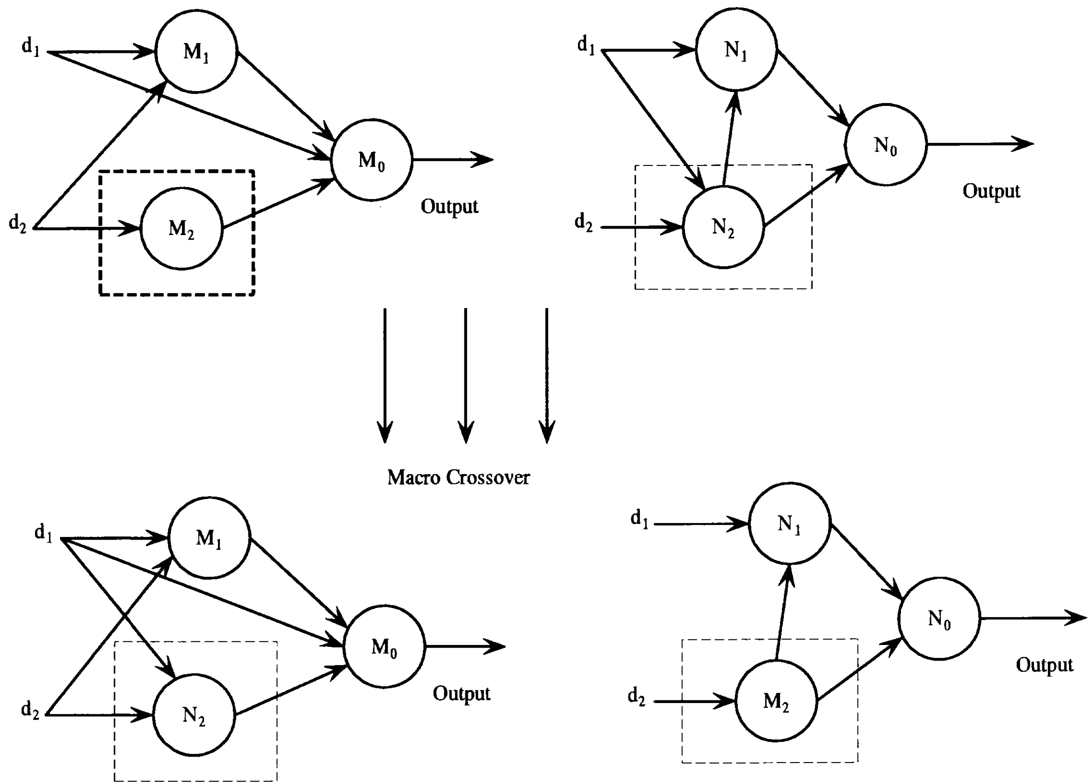

Figure 4.7: A typical crossover operation at EPNN individual level

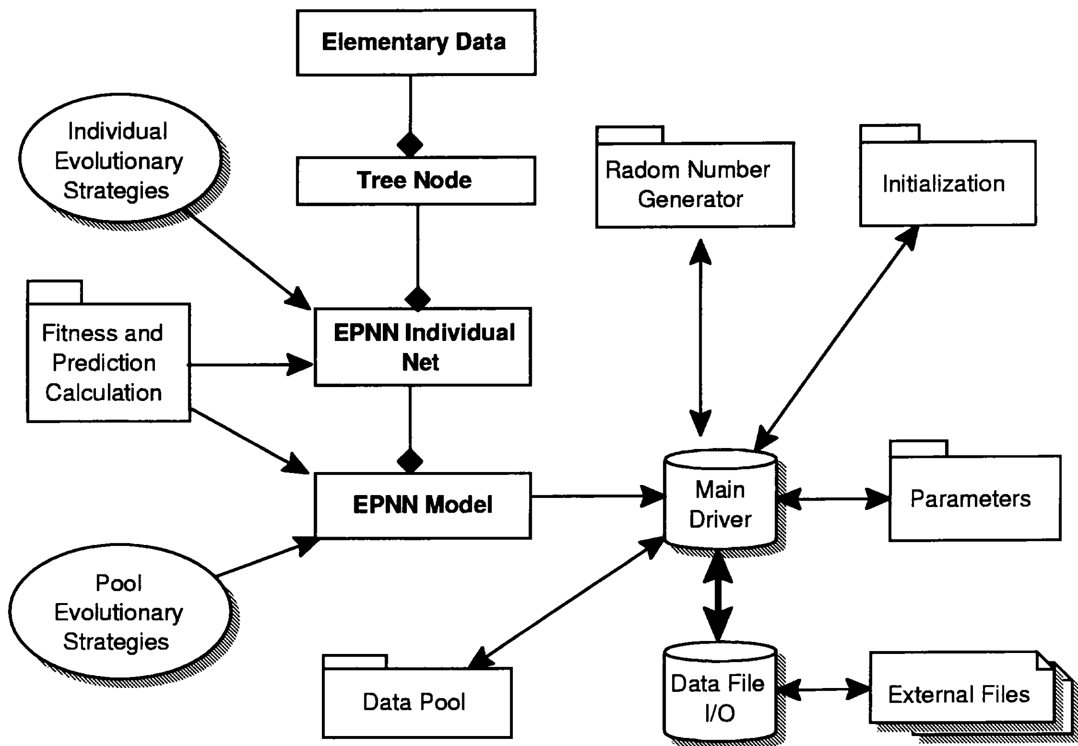

Figure 4.8: EPNN coding hierarchy scheme

CHAPTER 5

EXPERIMENTS AND RESULTS

5.1 Overview

In this chapter, three different process modeling examples are discussed. The first one is a dynamic neutralization process forecasting, the second one is prediction of aqueous two-phase system partitioning, and the last example is an application of EPNN to modeling the reduction of an existing biodegradation model.

As demonstrated in all of the following examples, EPNN performs better than or at least performances when compared to traditional thermodynamic or neural network models. EPNN also demonstrates the capability of producing symbolic formulae of the process and providing valuable information to establish potential theoretical models and optimization methods.

Furthermore, in the cases where traditional modeling parameters are hard to determine, EPNN can be utilized as an auxiliary efficient tool to produce equivalent empirical formulae for the target process. The last example, model reduction of a biodegradation process, demonstrates such potential of the EPNN approach.

5.2 Dynamic Chemical Reaction System

5.2.1 Introduction

In the process industry, it is necessary to predict several steps regarding the future of process outputs in many situations, for example, when calculating the control objective function in a model based predictive control. In many industrial processes,

there are certain variables, such as quality variables, which are difficult to measure. The values of these variables can be estimated from measured process variables by using a dynamic model of the process. In some cases, however, changes in the process inputs can affect the quality variables in the long run. Therefore, multiple step ahead prediction is desirable in such cases to improve the estimation performance. A multiple-step-ahead prediction model can be described as Equation 5.1.

$$\begin{aligned} \hat{y}(t) =& f(\hat{y}(t-1), \hat{y}(t-2), \ldots, \hat{y}(t-n), \\ & u(t-1), u(t-2), \ldots, u(t-m)) \end{aligned} \tag{5.1}$$

where the model predictions $\hat{y}(t-1)$ to $\hat{y}(t-n)$ are used as process outputs and $u(t-1)$ to $u(t-m)$ are process inputs.

A neutralization CSTR simulation example (Figure 5.1), which has been extensively investigated in the literature, such as the long-term prediction model by (Zhang et al., 1998a), is used to show that the EPNN approach can efficiently predict the highly nonlinear system for nearly chaotic data series. The dynamic thermodynamic model for the pH neutralization CSTR process can be found in the literature (McAvoy et al., 1972). There are two input streams into the continuous reactor, one is acetic acid of concentration C_a at a flow rate F_a and the other is sodium hydroxide of concentration C_b at a flow rate F_b. The acid input stream neutralizes the sodium hydroxide. From the titration curve of the process, it is indicated that the process is highly nonlinear around $pH \approx 7$. In the modeling process, the flowrate of sodium

hydroxide and the tank temperature are kept constant, while the flowrate of acetic acid is constantly disturbed due to random noise in the environment. The objective of EPNN modeling is to produce long-term time series forecasting of the pH values in the dynamic system only in relation to the input with disturbance, F_a, which is the input flowrate of acetic acid.

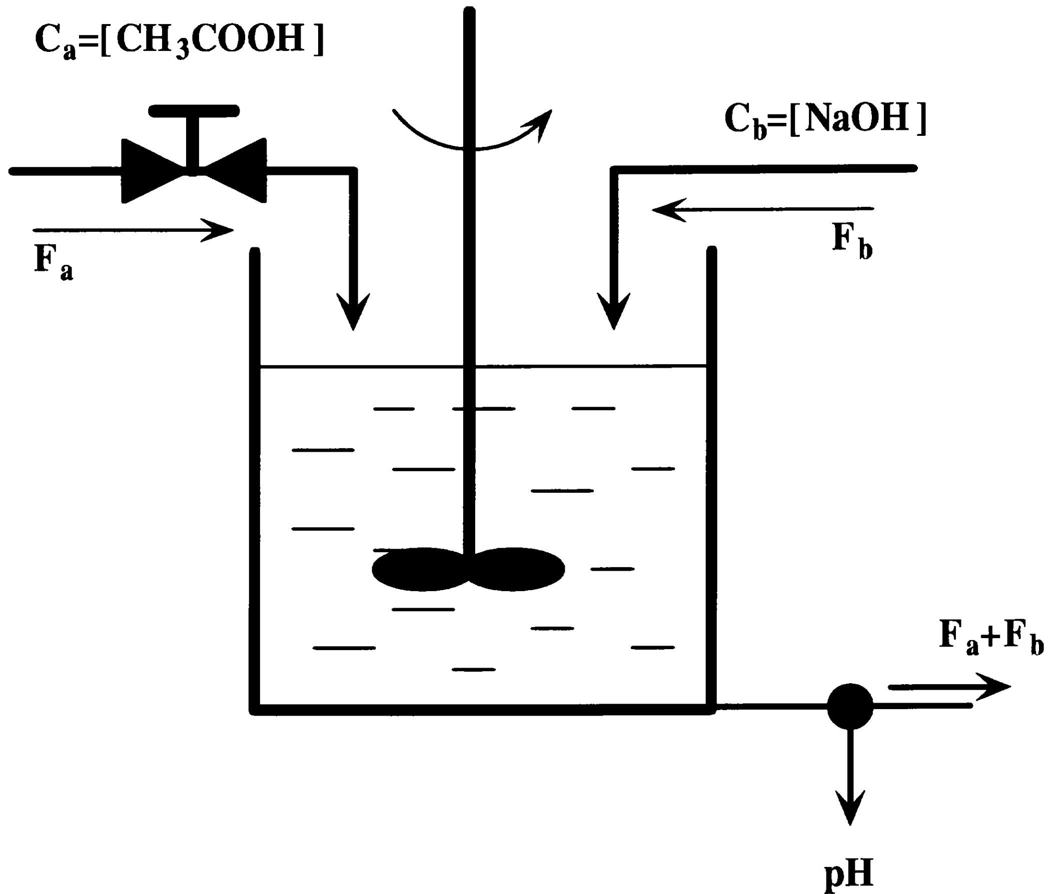

Figure 5.1: Neutralization CSTR reaction

In order to compare with the performance of an existing neural network, the complete data set that was used in a published paper (Zhang et al., 1998a) was acquired . A total of 600 pairs of data points was acquired. The first 300 pairs were used as the training set, while the remaining 300 pairs as the testing set.

The EPNN model was developed using the standard C/C^{++} computer language under Windows NT environment. A total of five nodes were used in the model, in which there are four intermediate nodes and one output node. The output variable is the predicted value of the current pH level at time t, while the two input variables are: the acetic acid flowrates at time $t-1$ and at time $t-2$. It is important that both the inputs and outputs are normalized to a standard value range of $[-1,1]$ by rescaling. The detailed scheme of the EPNN training algorithm is illustrated in Figure 4.4. The fitness in Figure 4.4 is calculated by the method introduced in the previous chapter. RMS in Equation 4.7 is the reciprocal of summed square errors(SSE) of normalized pH predictions of each individual in the EPNN population. In this work, the size of the gene pool is 150 individuals, while the size of the mutation pool is 80.

5.2.2 Results and Discussion

The first set of 300 data points was used in the training process. A total of 3000 generations was evolved before the training program was automatically stopped. The training process took about 6 hours on our computing platform (Pentium II 233MHz and Windows NT 4.0). Following the training process, the fittest individual network(the EPNN individual with the highest fitness function value) was selected as the solution. The second set of 300 data points was used in the prediction process. Prediction results of the solution are shown in Figure 5.2 , while the associated errors are shown in Figure 5.3. Both Figures 5.2 and 5.3 demonstrate a good approximation over most of the data. The spikes in Figure 5.3 are mainly caused by the uniformly

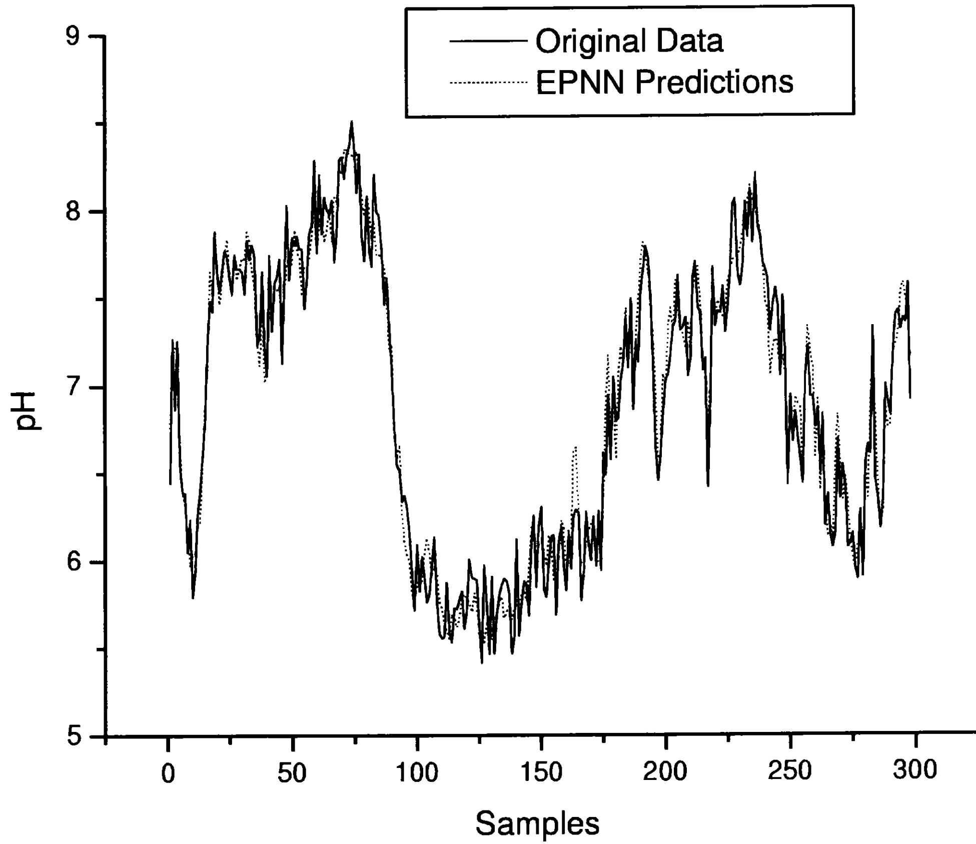

Figure 5.2: CSTR EPNN prediction results

distributed random noise in the range of (-0.3, 0.3) in the original experimental pH measurement. The shape of the error curve in Figure 5.3 is not uncommon in most time series forecasting applications (Zhang, 1994; Dorffner, 1996; Oliveira et al., 2000).

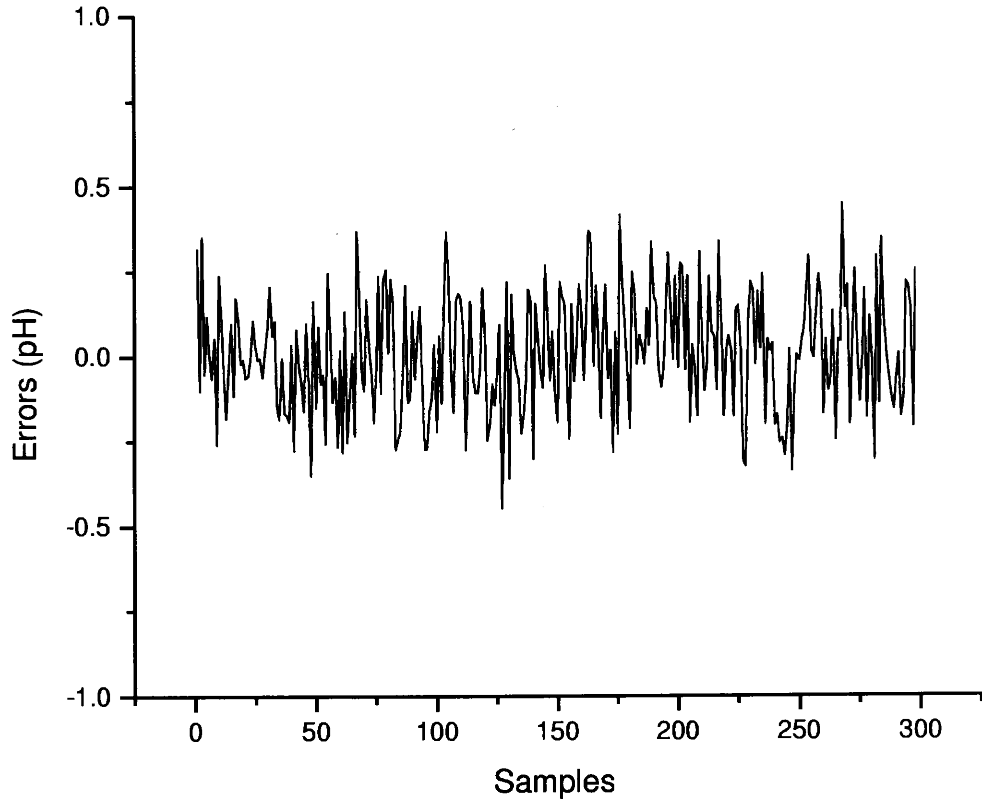

Figure 5.3: CSTR EPNN prediction errors

The evolved symbolic formula of the fittest individual is shown in the system of equations 5.2.

$$
\begin{aligned}
N_0 &= \mathcal{F}(N_1, N_2, N_3, d_1, d_2) \\
N_1 &= \exp(0.204195 \times N_0 \times N_1) \\
N_2 &= N_0 - 0.590388 \\
N_3 &= \exp\big(\ln(N_2 + N_4) \times 0.338454 + (\exp(N_0) \times N_4 - N_3) \times N_2\big) \\
&\quad + 3.263768 \times d_1 \\
N_4 &= 0.97810 \times N_4 + 0.644332 - \exp(1.990765 + d_2) \times d_1 \times 1.0012
\end{aligned} \tag{5.2}
$$

where N_0 is the output node — normalized pH value prediction, $\mathcal{F}$ is a complicated algebraic function shown in equation 5.3; N_1,N_2,N_3,N_4 are intermediate nodes; d_1,d_2 are the model input variables — normalized acetic acid flowrates at time $(t-1)$ and $(t-2)$, respectively.

$$\begin{aligned}\mathcal{F}(N_1, N_2, N_3, d_1, d_2) = & \{0.109183 - exp(0.157093/N_3)\} \\ & \times \left\{\frac{N_1 - 0.990306 \times exp(d_1)}{N_2 + d_2 \times (0.826854 - d_1)}\right\}\end{aligned} \tag{5.3}$$

The associated network structure of the system of equations 5.2 is illustrated in Figure 5.4. Unlike ANNs or ANNs with GAs, this model produces empirical equations for the given system. In addition, despite the random noise in the pH measurement, the system of equations 5.2 achieved an SSE of 8.8 while the traditional recurrent neural network has an SSE of 17.0 (Zhang et al., 1998a).

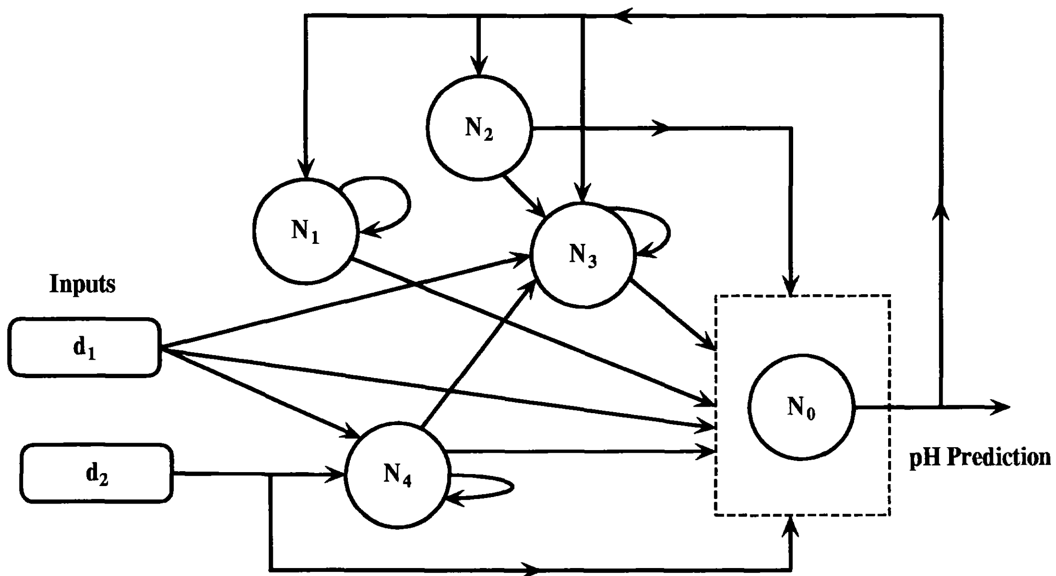

Figure 5.4: Evolved EPNN network for CSTR

One of the most important features during the evolution of the EPNN network is that in less than 10 generations the network can create feedback links between nodes, if necessary. These links represent the acetic acid flow rate at previous time intervals, such as $(t-3)$, $(t-4)$ even though the initialization of the EPNN network is only feedforward and has only two inputs: acetic acid flowrates at time $(t-1)$ and $(t-2)$. By contrast, the structure of recurrent neural network must be determined before training and cannot be modified during training (Zhang et al., 1998a). Consequently, it demands a significant amount of effort to determine the final network structure. Therefore, EPNN is more adaptable and efficient in network structure determination than traditional recurrent neural networks. The evolved feedback links are capable of modeling multi-step ahead (at least two steps ahead) prediction for the examined nonlinear dynamic system.

Furthermore, the final EPNN network has only 4 intermediate nodes(neurons) with performance of $SSE \leqslant 8.8$, while the traditional recurrent neural network had to utilize at least 14 neurons to achieve $SSE = 17.0$. Too many neurons can introduce overfitting of data and yield poor generalization of the system due to too many links between intermediate nodes in the network (Zhang et al., 1998a; Doherty et al., 1997).

In addition to producing improved results, EPNN also evolved empirical equations (Equation 5.2). The evolved formulae not only provide information for further theoretical study, but can also be used as empirical equations for process optimization and dynamic control. Therefore, EPNN is fundamentally different from those

methods that combine ANN and GA in chemical process modeling (Zhao et al., 2000; Gao et al., 1999).

From the above investigated case, it is clear that the EPNN approach to modeling complex, dynamic system behaviors is capable of recognizing dynamics as well as efficiently constructing empirical models for the process. Feedback links in EPNN networks can be formed through training(evolution) to perform multi-step ahead prediction for nonlinear systems. Furthermore, unlike those applications combining ANNs and GAs, symbolic formulas can be extracted from EPNN modeling results for further theoretical analysis and process optimization and control.

5.3 Aqueous Two Phase Extraction System

In this section, EPNN is applied to modeling a complex aqueous two phase extraction system. The system, polyethylene glycol (PEG)/potassium phosphate/water at pH=7 was selected to demonstrate the performance of the EPNN model. The results were compared favorably to a traditional neural network modeling approach and the experimental data set. Seven distinct data sets of varying PEG molecular weight were used in this work. Of the seven, five were used for training while the remaining two were employed as the test cases. Following the training, a networked symbolic equation system evolved, which, in addition to reproducing the data, can also be used to improve understanding of the phase diagram through the discovered parameters.

5.3.1 Introduction

In macromolecule and biological material processing industries, aqueous two-phase systems (ATPS) have been applied in separation and purification of various substances. The primary advantages of ATPS handling of materials are biocompatible environments, economical operation and scaleup and adjustable factors to manipulate target product partitioning (Albertsson, 1971). However, while there are many factors available to manipulate the partition, it is difficult to correlate and model the phase equilibrium and partition due to the highly complicated interactions between those factors (Kan and Lee, 1996).

A number of thermodynamic models have been developed and applied to describe the mechanism of phase separation. The Flory and Huggins theory (Flory et al., 1964) has been applied successfully in modeling the polyethylene glycol (PEG)/dextran systems (Gustaffsson et al., 1986). Later, models based on statistical thermodynamics, such as UNIQUAC or UNIFAC (Abrams and Prausnitz, 1975; Fredenslund et al., 1975) were developed and applied to correlate and extrapolate phase diagrams with system parameters by fitting other similar phase equilibrium data (Gao et al., 1991). However, there is a common drawback in predicting phase diagrams using the above models. Generally speaking, a thermodynamic model requires a set of specific parameters which are only valid exclusively for a particular system. For example, the model of phase equilibrium with parameters measured from the Dextran 500/PEG 1000 system may not be able to forecast phase equilibrium for a Dextran 500/PEG 3400 system. Therefore, in order to apply those

thermodynamic models in laboratory or industrial process design, a huge parameter database for numerous systems is required. However, the special equipment to determine these parameters may not be available in every laboratory or factory (Kan and Lee, 1996).

Since the mid 1990s, in order to overcome the limitations of traditional thermodynamic models, there have been a number of efforts of applying neural networks to predict phase equilibrium data, for systems such as PEG/potassium phosphate/water (Kan and Lee, 1996). The advantages of these neural network approaches are stated as nonparametric models (Tsoukalas and Uhrig, 1997; Caudill and Butler, 1992). Structurally, they are adjustable and adaptable for different phase systems without re-evaluation of thermodynamic parameters. However, there are limitations to these approaches. The most cited one is the nonparametric nature or "blackbox" operation, which makes it difficult to establish efficient structures of neural network and training methods (Doherty et al., 1997). While the trained model has performed very well on the testing data sets, it does not provide information about the potential mechanism of the system. Therefore, EPNN is applied to modeling the two-phase extraction system.

5.3.2 Data Preprocessing

The phase equilibrium data of the PEG / potassium phosphate aqueous two-phase system (pH 7) were obtained from experiments reported in the literature (Lei et al., 1990; Kan and Lee, 1996). Figure 5.5 illustrates the extraction process. The total

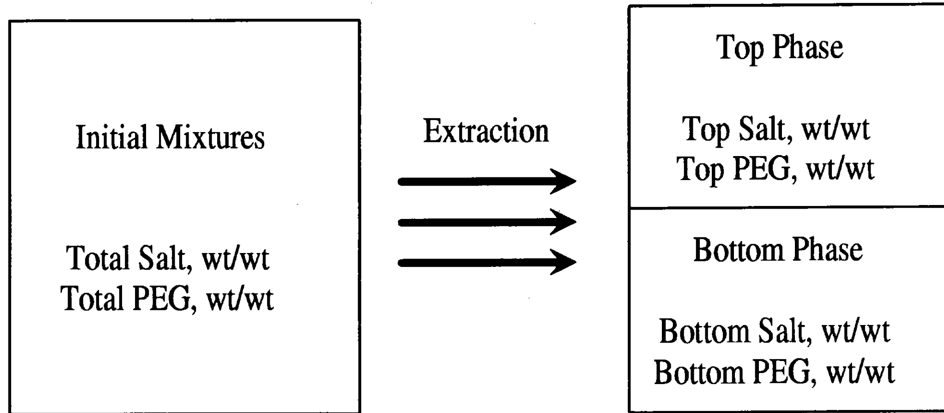

Figure 5.5: Two phase extraction system for PEG

weight proportions of PEG and potassium phosphate in the two-phase system are two inputs to the EPNN system. In addition, the normalized PEG molecular weight(MW) was utilized as a third input. Because the molecular weight of PEG commonly applied for ATPS is below 50,000, the normalized MW is expressed in Equation 5.4, which is adopted from the literature (Kan and Lee, 1996).

$$P_{MW} = \frac{\log(MW/100)}{log(500)} \qquad (MW < 50,000) \tag{5.4}$$

In the two-phase extraction system, the size of an EPNN population is 100 and the size of mutation pool is 50. The maximum depth of any formula tree is restricted to 100. The function set consists of the following operators: $\{ +, -, \times, \div,$ *ln* and *exp* $\}$. There are four intermediate nodes: N_4, N_5, N_6 and N_7. The number of output nodes is 4, as shown in Figure 5.6. The inputs to the model are d_1, normalized PEG molecular weight, d_2, total concentration of potassium phosphate and d_3, total concentration of PEG.

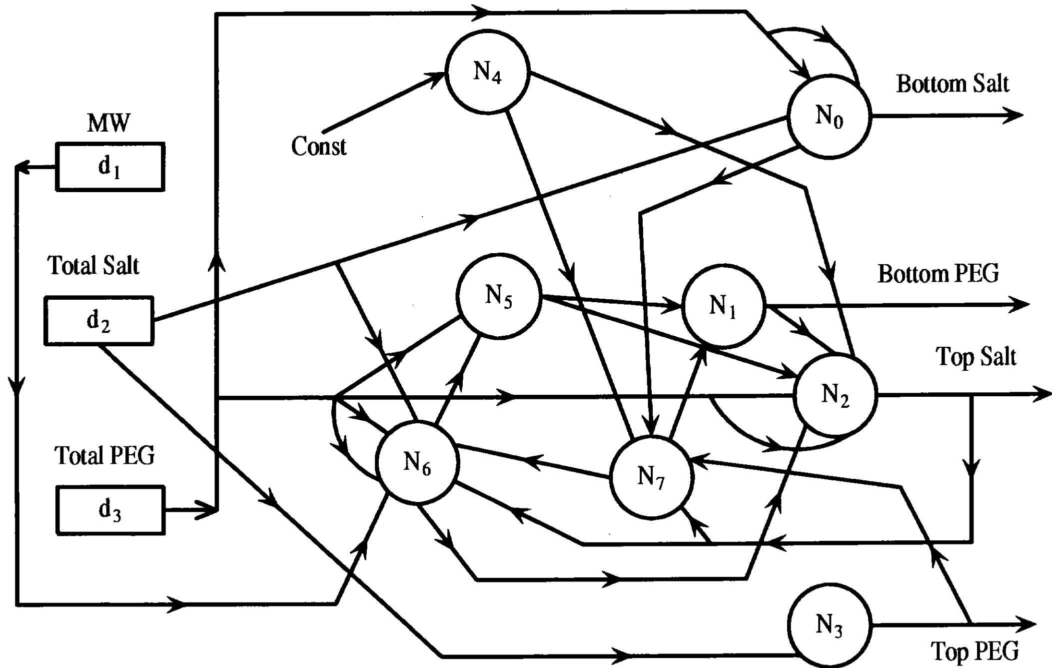

Figure 5.6: Trained EPNN network structure for PEG system

During the training step of the EPNN model, all the data from PEG 600, 1,500, 3,400, 8,000 and 20,000 are used to train the model until satisfactory prediction precision is reached. Following the training, the prediction of the model for both interpolation and extrapolation was tested. PEG 1000 and PEG 400 cases are used in this step. Each data set consists of 8 data points. A total of 40 data points were used for the EPNN system training in a single run. The remaining 16 data points were used to test the EPNN system performance in a different run.

5.3.3 Results and Discussion

During the simulation, 20,000 generations were evolved before the training program automatically stopped. Then the fittest individual network was selected as the

solution (Figure 5.6). The corresponding symbolic networked formulae of the fittest EPNN network are:

$$
\begin{aligned}
N_0 &= 1.08305 \times d_2 + \frac{d_3^2}{0.287544} \\
N_1 &= \frac{N_7 \times 0.1}{\exp(1.20421 \times N_5)} \\
N_2 &= f_1(N_1, N_4, N_5, N_6, d_3) \\
N_3 &= \frac{d_3 \times 1.4497}{d_2 + 0.64995} \\
N_4 &= 0.729241 \\
N_5 &= N_6 + \frac{d_3}{0.154417} - 0.377927 \\
N_6 &= \left\{ \frac{\exp\left\{(N_2 + d_3) \times 1.82215 + \ln(d_3/d_2)\right\}}{1.95362} \right\} \times d_1 \\
N_7 &= \exp\left\{ \frac{N_0 \times 0.913581}{\exp\left(\frac{(N_6 - 0.410872)\times(N_2 - 0.626065)}{N_3}\right)} \frac{1}{N_6} \right\}
\end{aligned}
\qquad (5.5)
$$

where N_0, N_1, N_2 and N_3 are the output nodes. f_1 is:

$$
\begin{aligned}
f_1(N_1, N_4, N_5, N_6, d_3) = 0.1 \times N_4 \times ln(exp(f_2) \\
+ (0.88749 - d_3) \times 0.134139)
\end{aligned}
$$

and f_2 is:

$$
\begin{aligned}
f_2 = (0.812427 \times exp\{(N_6 - 0.981641 \times exp(0.7405 + N_1) + d_3)\} \\
+ 0.917017)/(d_3 - exp(N_5) \times ln(N_4))
\end{aligned}
$$

From Equation 5.5 and Figure 5.6 it can be seen that the nodes in the network are interconnected with each other by back-links (feedbacks). Those back-links were formed automatically during the training step.

The degree of complexity of the evolved network is related to the precision of the given data. As indicated in the experimental data (Lei et al., 1990; Kan and Lee, 1996), bottom PEG(N_1) and top salt (N_2) concentrations are minor components in their respective phases. This may result in a higher level of error/uncertainty due to the magnitude of these concentrations (Lei et al., 1990; Kan and Lee, 1996). Therefore the EPNN network, in striving to maintain a constant error, forms a more complicated structure.

In this work the intermediate nodes are not germane to the interpretation of the results.

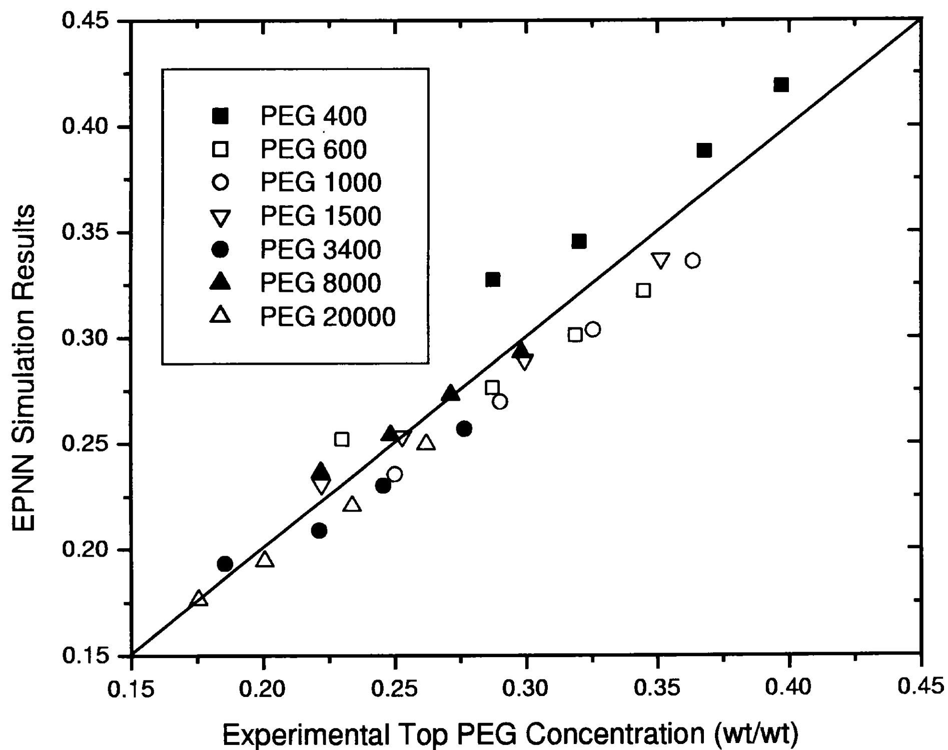

Figure 5.7: Top phase PEG concentration study

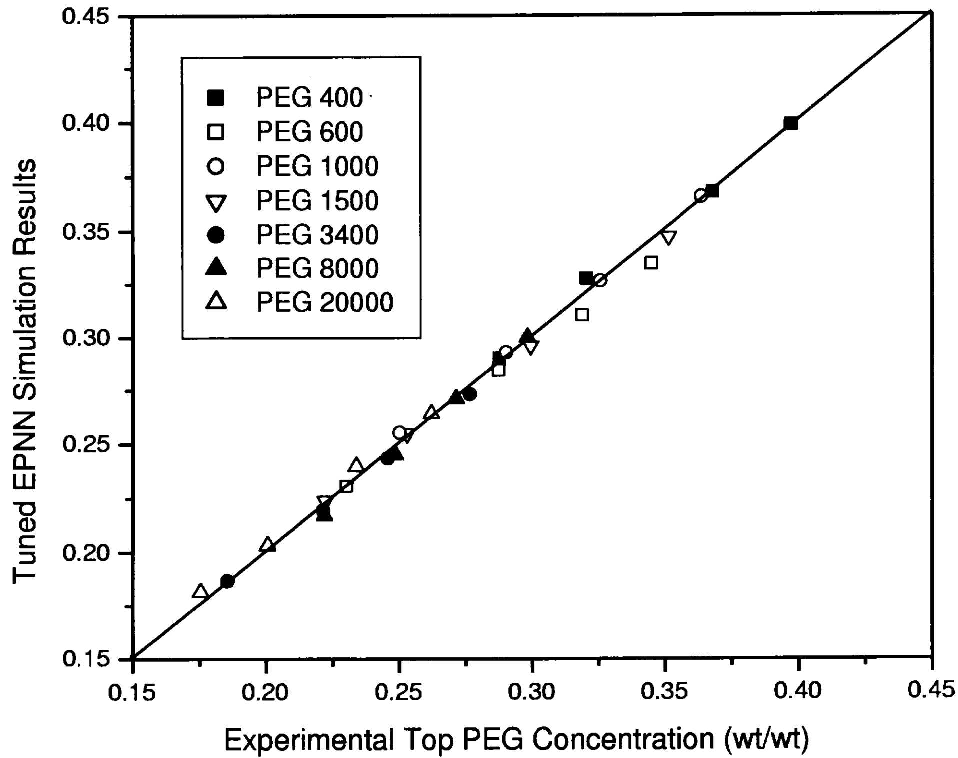

Figure 5.8: Tuned top phase PEG predictions

Examination of Equation 5.5 reveals some interesting and useful information about the PEG/potassium phosphate/water system:

- Concentration of PEG in top phase (N_3) is approximately a function of total PEG concentration (d_3) and total salt concentration (d_2).
- Concentration of salt in bottom phase (N_0) is approximately independent of the molecular weight of PEG (d_1) and thus only depends on the total concentration of PEG (d_3) and potassium phosphate (d_2).

- Concentration of the minor component in each phase (N_1 and N_2) is more complicated and interconnected than the major component concentration (N_0 and N_3) in their respective phases.

Albertsson (1971) showed that for a constant pH (pH=7), and a proper experimental salt concentration range, the PEG concentration in both phases remain constant regardless of the PEG molecular weight. According to Figure 5.7, the experimental results of Albertsson are approximately confirmed by the EPNN model. In particular, it is shown that PEG concentration in the top phase is dependent on total PEG concentration. However, the top phase PEG concentration can be more accurately predicted (Figure 5.8) by tuning the EPNN model with different initial system conditions. Now, the top phase PEG concentration, N_3, depends on all three variables:

$$\begin{aligned} N_3 =& (1.196 \times d_2 + 1.149 \times d_3 + 2.0149 \times d_1 \times d_2 \\ &+ 0.1142 \times d_1 \times d_3 - 0.1908636) \end{aligned} \tag{5.6}$$

During the tuning step, only one total node was used, which is the output node. Only 50 generations are evolved to produce the improved approximation. The EPNN training results can be tuned with only minimum changes in the initial conditions. Therefore, unlike traditional neural networks, which demand a significant effort to change network structures before tuning (Doherty et al., 1997), the network structure of EPNN is more flexible and adaptable through tuning. This is an advantage of the EPNN's polymorphic nature over traditional neural networks.

As indicated in the experimental data (Lei et al., 1990), the product of $d_1 \times d_2$ or $d_1 \times d_3$ is really small and only second-order correlated to N_3, while d_2 and d_3 are both larger and first-order correlated to N_3. Therefore according to Equation 5.6, N_3 is less sensitive to d_1 (MW of PEG). The observed result of independence for N_3 (top PEG concentration) still holds true in the approximation.

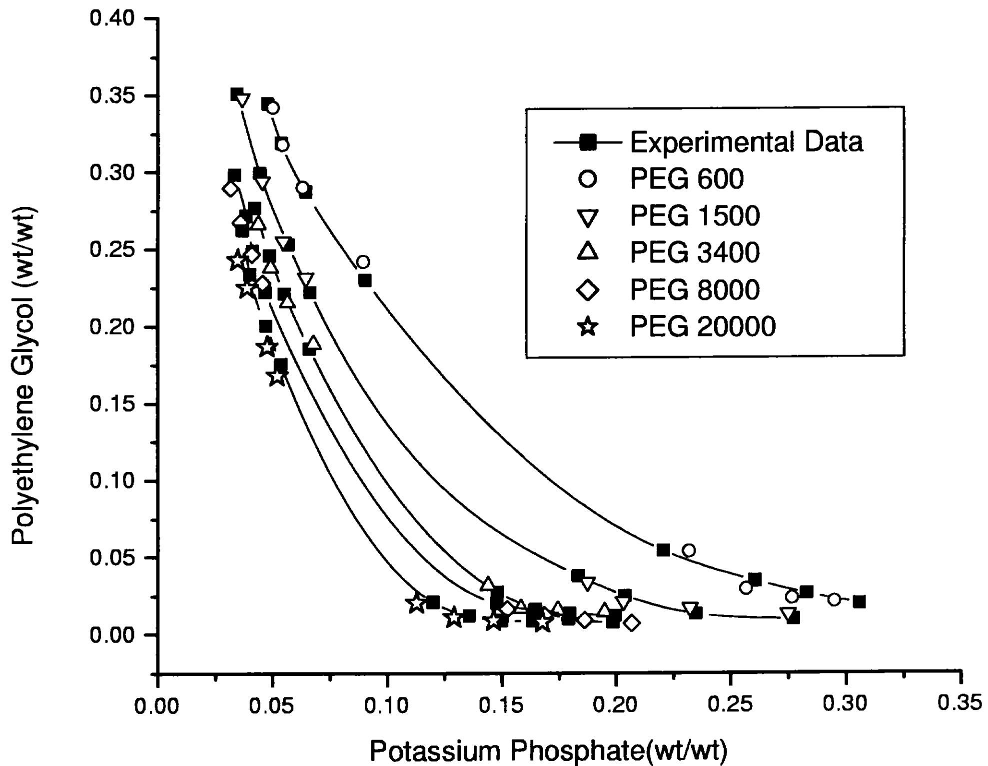

Figure 5.9: Phase diagram (training results)

Figure 5.9 shows the training results after the tuning. The top left part of the figure shows the concentrations in the top phase, and the bottom right part shows the concentrations in the bottom phase. Figure 5.10 shows the associated training errors. Both Figure 5.9 and Figure 5.10 show good agreement with the experimental

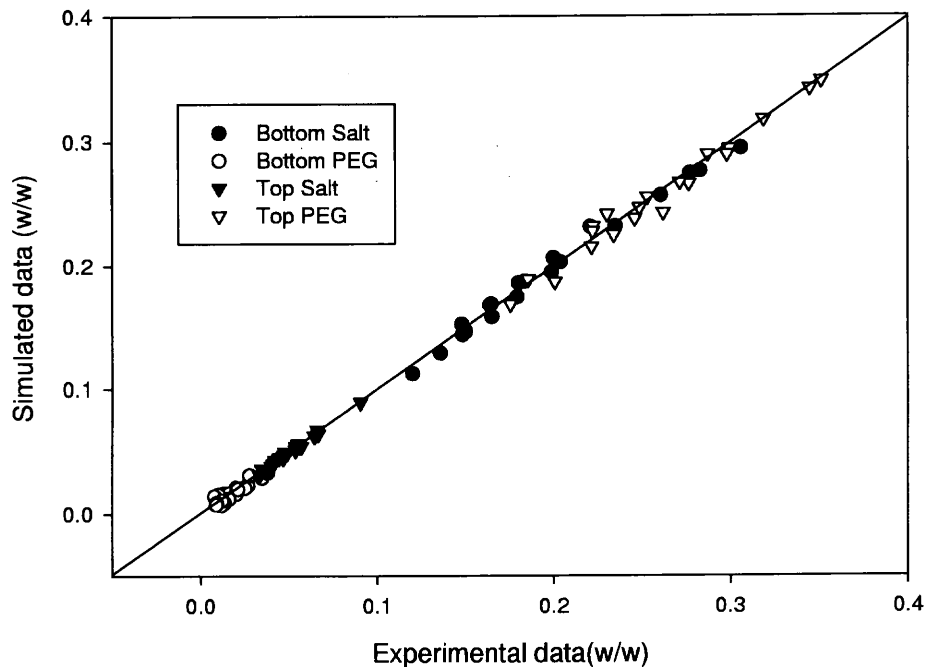

Figure 5.10: EPNN training errors

data (Lei et al., 1990) over the five training data sets.

Figure 5.11 shows the prediction results using the tuned network structure. As in Figure 5.9, the top left part of the figure shows the concentrations in the top phase, and the bottom right part shows the concentrations in the bottom phase. Figure 5.12 shows the associated prediction errors. Again, both Figure 5.11 and Figure 5.12 show good agreement with the experimental data over the two testing data sets. The standard deviations of EPNN prediction are 0.0114 wt/wt (PEG 400) for extrapolation and 0.00302 wt/wt (PEG 1000) for interpolation, while the traditional neural network approach errors are, respectively, 0.0118 wt/wt (PEG 400)

and 0.0085 wt/wt (PEG 1000) (Kan and Lee, 1996). Detailed prediction results are listed in Table 5.1 – Table 5.4.

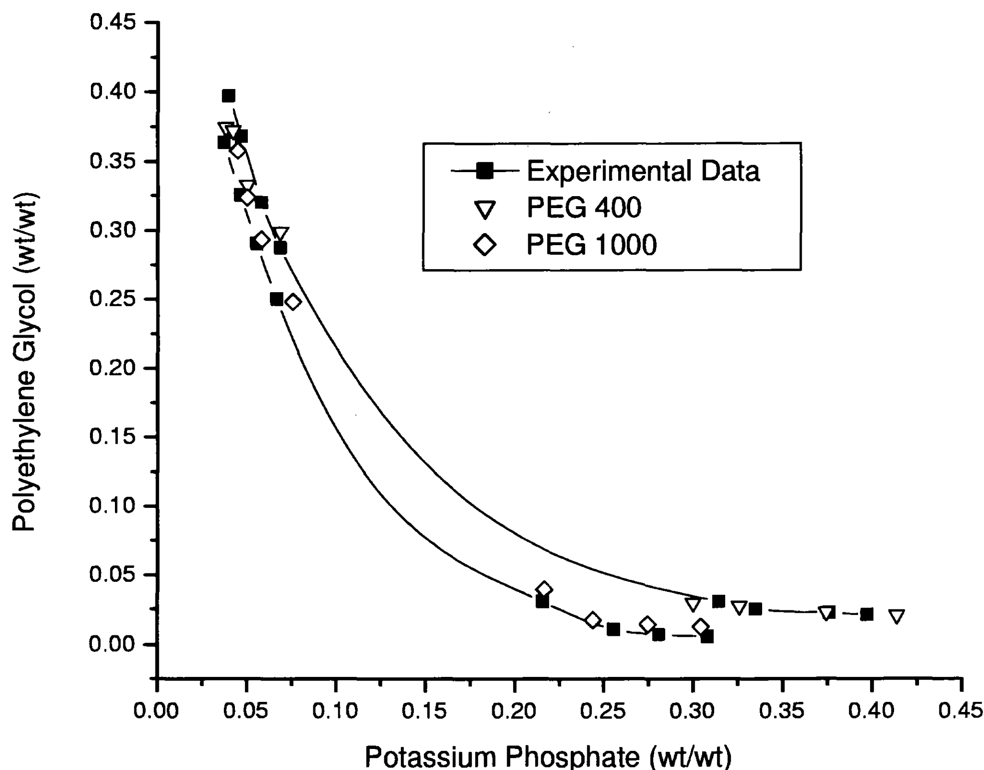

Figure 5.11: Phase diagram (prediction results)

In addition to producing better results, EPNN also evolved empirical formulae (Equations 5.5 and 5.6), which can be applied over the whole range of the studied system. The formulae produced by EPNN not only provide valuable information for further theoretical study, but can also be used as empirical equations in ATPS process optimizations (Huenupi et al., 1999).

Due to the polymorphic and evolutionary nature of the EPNN system, the initially randomized values of constants in EPNN networks will converge to the

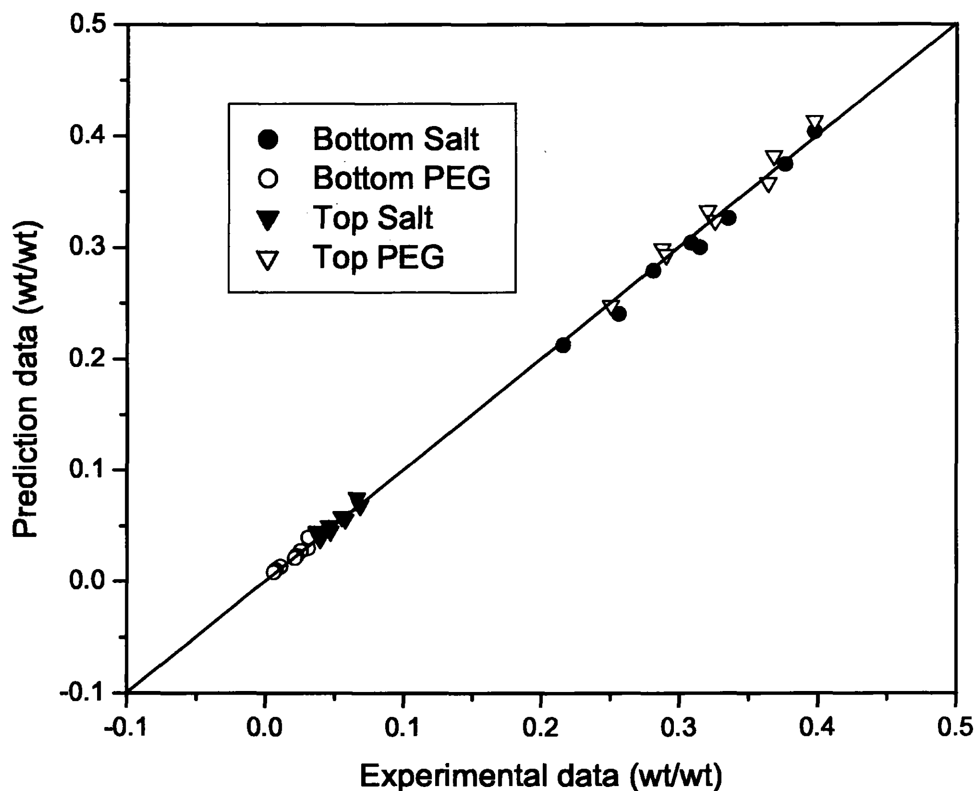

Figure 5.12: EPNN prediction errors

same or similar forms of functions in separate runs until the training process ends. Therefore, the EPNN system is not sensitive to differences in initial constant values of the EPNN population. However, if there is significantly larger noise in one or more data sets in the whole data composition, the EPNN system will probably fail to converge to a satisfactory level of prediction on these data sets. Therefore, the precision of the training data is critical to successful and efficient modeling using EPNN.

Table 5.1: Comparisons between EPNN prediction and experimental data for PEG 400 in bottom phase

Potassium Concentrations		PEG Concentrations	
Experimental	EPNN	Experimental	EPNN
0.3145	0.30019	0.0305	0.02982
0.3351	0.32607	0.0254	0.02709
0.3761	0.37449	0.0224	0.02305
0.3973	0.40367	0.0212	0.02095

Table 5.2: Comparisons between EPNN prediction and experimental data for PEG 400 in top phase

Potassium Concentrations		PEG Concentrations	
Experimental	EPNN	Experimental	EPNN
0.0685	0.0685	0.2875	0.29905
0.0578	0.05605	0.3202	0.33274
0.0469	0.04498	0.3679	0.37224
0.0395	0.03805	0.3973	0.38456

Table 5.3: Comparisons between EPNN prediction and experimental data for PEG 1000 in bottom phase

Potassium Concentrations		PEG Concentrations	
Experimental	EPNN	Experimental	EPNN
0.2156	0.21639	0.0308	0.03959
0.2556	0.2439	0.0108	0.01362
0.2808	0.27463	0.0071	0.01053
0.3081	0.30447	0.0056	0.00854

Table 5.4: Comparisons between EPNN prediction and experimental data for PEG 1000 in top phase

Potassium Concentrations		PEG Concentrations	
Experimental	EPNN	Experimental	EPNN
0.0664	0.07545	0.2502	0.24818
0.0553	0.05709	0.2902	0.29344
0.046	0.04979	0.3256	0.32371
0.037	0.04462	0.3637	0.35769

5.4 Bioremediation Model Reduction

5.4.1 Introduction

As the worldwide concern for the protection of the environment grows, more people are seeking and developing ways to eliminate the contaminants from more than two hundred years of industrial activities. One of the promising techniques to eliminate organic contaminants is biodegradation. With the increasing applications of biodegradation in waste treatment, better modeling and better understanding of the ongoing biodegradation process is also increasingly demanded. However, due to the nature of complex mechanism of such processes, it is difficult to develop a complete and reliable, structured model using traditional methods without many unsuccessful modeling trials and arbitrary assumptions. Such limitations can significantly decrease the applicability and reliability of existing models. Therefore, it is desirable to develop alternative ways to model biodegradation processes.

The focus in this section will be on the model reduction of an existing in-situ bioremediation model using EPNN techniques. The original model and processes can be found in the literature (Mandal, 1998). The original model was developed for biodegradation in a soil batch bioreactor.

In order to model the process, the original approach made simplifying assumptions and divided the process into two levels of material balances: aggregate phase and mobile phase. For the aggregate phase, the material balance of the biodegradable component in a different segment of the spherical aggregates is represented in Equation 5.7:

$$\frac{\partial C_a}{\partial t} = \frac{D_{se}}{r^2}\left[\frac{\partial}{\partial r}\left(r^2\frac{\partial C_a}{\partial r}\right)\right] - B_d - A_d \tag{5.7}$$

The first term on the right represents the intra-aggregate diffusion, the second term represents the rate of biodegradation and the third term is the adsorption rate. The biodegradation rate (B_d) and the associated biomass growth rate are given by the equations 5.8 and 5.9, respectively (Mandal, 1998):

$$B_d = -\frac{\partial C_a}{\partial t} = \frac{b}{Y}\left[\frac{\hat{\mu} C_a}{K_s + C_a + \frac{C_a^2}{K_i}}\right] \tag{5.8}$$

$$\frac{\partial b}{\partial t} = \left[\frac{\hat{\mu} C_a}{K_s + C_a + \frac{C_a^2}{K_i}}\right] \cdot b - (\hat{\mu_c}) \cdot b \tag{5.9}$$

Where C_a is the 2-CP concentration, b is the concentration of biomass, $\hat{\mu}$ is the specific growth rate constant in the Andrews model and $\hat{\mu_c}$ is the specific death rate constant. The specific growth rate was described by an inhibitory biokinetic model (Andrews model). One of the major problems in the original model is that it is extremely difficult to determine the exact value of the parameter ($\hat{\mu_c}$) in the aggregate phase model from experimental data. Therefore, arbitrary assumptions about $\hat{\mu_c}$ are needed on a limited number of available values in the literature during the parameter determination step as well as the final model verification step (Mandal, 1998).

Due to the arbitrary assumption on the modeling parameters, the original model fits poorly with the experimental data in some cases, although several trials had been made to guess an approximate value of the parameter. Therefore, in

order to avoid the above arbitrary assumptions in parameter determination, EPNN is applied in the biodegradation process modeling as an alternative and efficient way to reduce the existing model to the forms without such arbitrary assumptions.

5.4.2 Data Processing and Results

The experimental data are obtained from the literature, which are from the biodegradation experiments of 2-chlorophenol by a pure culture (*Pseudomonas pickettii*) in a jacketed batch reactor (Mandal, 1998). Prio to applying EPNN to reduce this biodegradation model, the overall yield coefficient was assumed constant. Therefore the biomass concentration growth rates should be dependent only on their initial concentrations of biomass and 2-CP and the time of the process under the same experimental conditions (constant temperature and controlled O_2 concentrations). This is a reasonable assumption as made by Mandal (1998).

Under the above simplifying assumptions, empirical symbolic models can be extracted from EPNN modeling of the experimental biomass growth curves. In the modeling process, the input set of EPNN is: the current time of the process relative to the initial starting time; the initial concentration of 2-chlorophenol (2-CP) and the initial concentration of biomass. The total nodes(neurons) of each individual EPNN net is set to two, which are both the output nodes: the prediction of biomass concentration and the prediction of 2-CP concentration at the present time. The function set is: $\{+, -, \times, \div, \exp \text{ and } \ln\}$. The gene pool size was designed as 1500 and the mutation pool size was 1000. The maximum depth of formula tree is restricted

to 100. In order to simplify the procedure, the comprehensive fitness function is set to the combined mean squared error in Equation 5.10.

$$Fitness = \sqrt{\mathcal{R}_1 \times \mathcal{R}_2} \tag{5.10}$$

where $\mathcal{R}_1$ and $\mathcal{R}_2$ are the mean squared error of prediction of biomass concentration and the mean squared error of the prediction of 2-CP concentration, respectively.

All the 15 sets of experimental data were used to train the EPNN system in order to get maximum coverage of all possible data ranges. The experimental data were randomly mixed together prior to model training. Furthermore, statistical "significance test" (Jonathan, 1991) was utilized during the training process to avoid over-fitting data. The basic algorithm of significance test is to compare standard deviation for the target variable, obtained in the training process performed for the real data set, to standard deviations obtained using the same evolved formulae for artificially generated data sets with random permutations of the values of the target attribute between different records. If the latter is significant larger than the former one, then the evolved EPNN formulae has not over-fitted the real data set. A total of 6,727 generations evolved before the EPNN system reached a satisfactory solution. The final evolved symbolic system of equations extracted from the fittest individual EPNN net is listed in Equation 5.11 and the associated network structure is shown

in Figure 5.13.

$$
\begin{aligned}
N_0 =& 5.816 - 1.3458 \cdot d_1 - 0.56717 \cdot d_2 + 0.08385 \cdot d_3 + 0.0939 \cdot d_1^2 + 0.09614 \cdot d_2^2 + \\
& 0.30367 \cdot d_1 d_2 - 0.013 \cdot d_1 d_3 - 0.01196 \cdot d_2 d_3 \\
& + 0.53622 \cdot \exp(0.123 \cdot d_1) \\
N_1 =& 1.3534 + 1.5214 \cdot d_1 - 0.888 \cdot d_2 + 1.0454 \cdot d_3 - 0.1713 \cdot d_1^2 + 0.06317 \cdot d_2^2 \\
& - 0.44477 \cdot d_1 d_2 - 0.03262 \cdot d_1 d_3 - 0.00613 \cdot d_2 d_3 \\
& + 0.1436 \cdot \ln(0.4317 \cdot d_1 + 0.9565 \cdot d_3) \qquad (5.11)
\end{aligned}
$$

where N_1 and N_0 are the output nodes — prediction of biomass concentration and prediction of 2-CP concentration, respectively. d_1,d_2, d_3 are the model input variables — time, initial biomass concentration and initial 2-CP concentration, respectively. It has to be pointed out that the values of N_0 and N_1 are bounded to non-negative numbers by the boundary condition specified in Equation 5.12. When time (d_1) is larger than the boundary upper limit, the value of N_0 will be artificially set to zero and N_1 will be set to remain the maxima value reached at the boundary upper limit value of d_1.

$$
d_1 \leqslant f(d_2, d_3) \qquad (5.12)
$$

where $f(d_2, d_3)$ is an implicit function defined in Equation 5.13:

$$
\begin{aligned}
0 =& 5.816 - 1.3458 \cdot f - 0.56717 \cdot d_2 + 0.08385 \cdot d_3 + 0.0939 \cdot f^2 + 0.09614 \cdot d_2^2 + \\
& 0.30367 \cdot f \cdot d_2 - 0.013 \cdot f \cdot d_3 - 0.01196 \cdot d_2 \cdot d_3 \\
& + 0.53622 \cdot \exp(0.123 \cdot f) \qquad (5.13)
\end{aligned}
$$

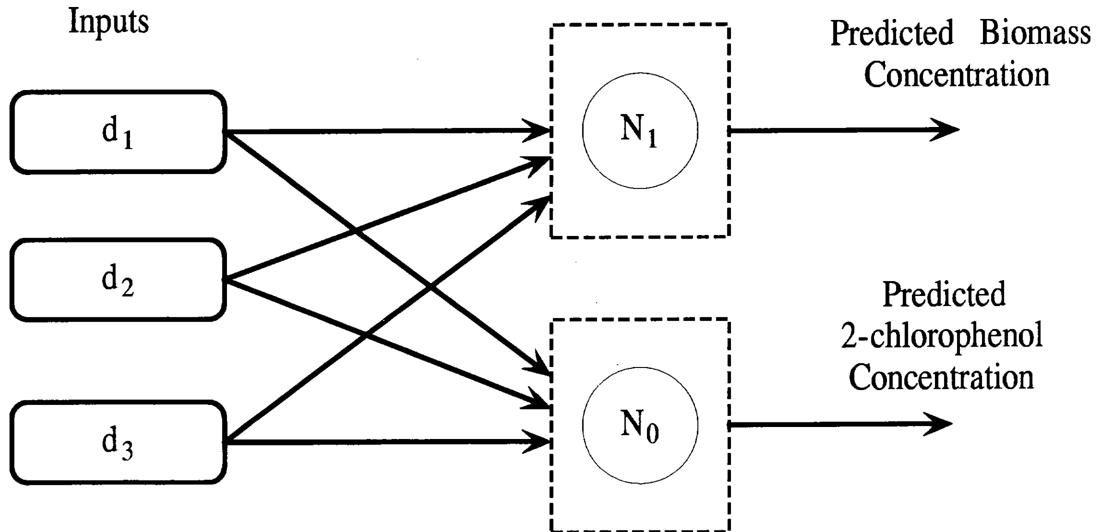

Figure 5.13: Evolved EPNN structure for biodegradation

Figure 5.14 and Figure 5.15 show an example of the training results of EPNN compared to the fitting results of the original model using experiment K-13. Table 5.5 shows the corresponding comparisons between EPNN prediction results and the original model for K-13. Because the original dissertation (Mandel, 1998) does not include the simulated data sheet, the simulated data of the Andrews based model were obtained using the *adaptive stepsize controlled Runge-Kutta* method, which can be found in the literature (Press et al., 1993). The mean square errors of EPNN prediction of biomass concentrations and 2-CP concentrations for K-13 are 0.3375 and 1.67, respectively, while the original model prediction for K-13 has mean square errors of 5.89 and 10.95. Besides K-13, for all other experimental data sets, EPNN achieves much better fitness than the original Andrews equation based model.

5.4.3 Discussion

From the above results of the EPNN prediction, it is shown that EPNN can produce a more accurate empirical model for the studied biodegradation process. In addition, unlike the original Andrews equation based growth rate model, the EPNN approach does not require any arbitrary assumptions on modeling parameters, or any trials before *guessing* a value of the model parameters. The EPNN can automatically establish empirical symbolic relationships between initial concentrations and the current concentrations through the training process.

The discovered relationship equations (Equation 5.11) can be used as a substitute for the Andrews equation based growth rate model in the original modeling system. Therefore, the biodegradation rate (B_d) in Equation 5.7 can be replaced with EPNN empirical equations (Equation 5.11). In such a modified model, it is thus not necessary to make trials and use *guessing* about the value of $\hat{\mu_c}$ in further calculations. However, asymptotic behavior cannot be predicted over time and must be monitored (such as artificial cut off negative values) in order to avoid spurious results obtainable about deterministic modeling.

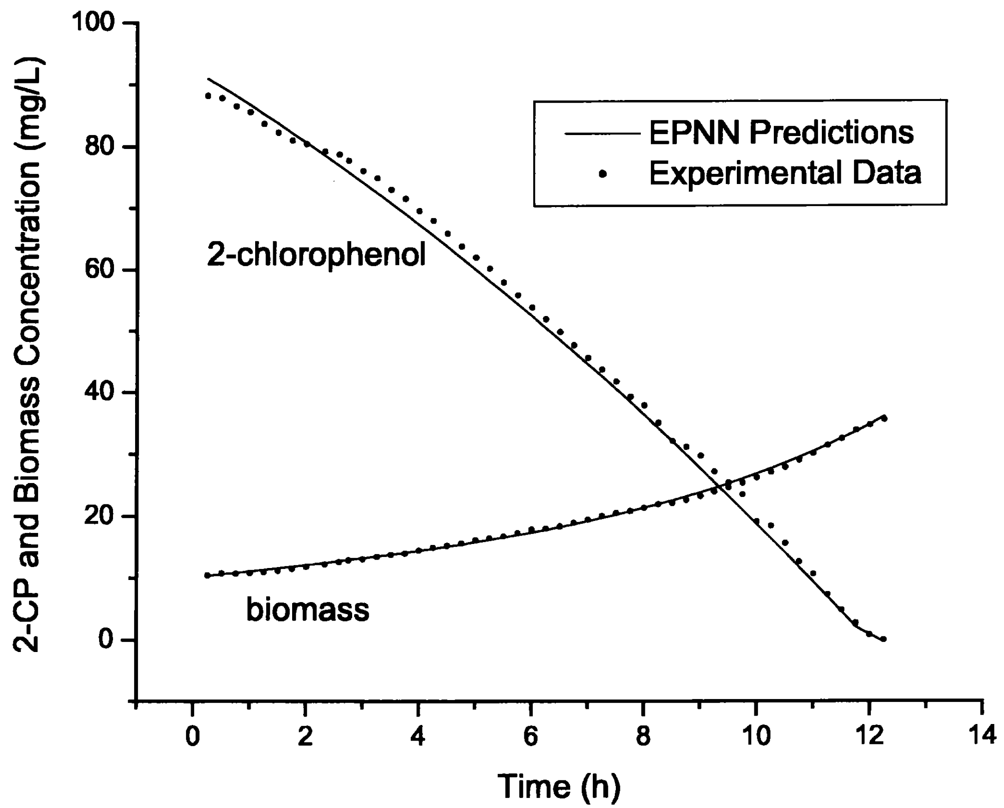

Figure 5.14: Comparison between experimental data and EPNN prediction of concentration profiles for 2-chlorophenol and biomass in experiment K-13

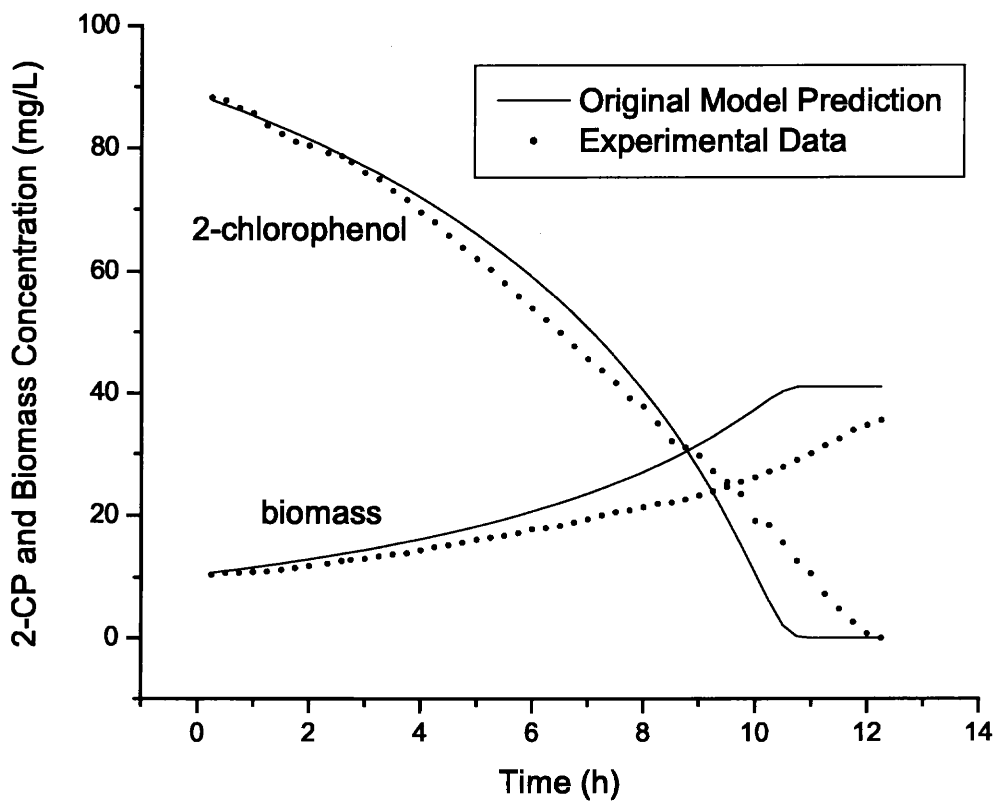

Figure 5.15: Comparison between experimental data and the original Andrews model prediction of concentration profiles for 2-chlorophenol and biomass in experiment K-13

Table 5.5: Prediction comparison for experiment K-13

Time(h)	Biomass (Exp)	2-CP (Exp)	Biomass (EPNN)	2-CP (EPNN)	Biomass (Andew)	2-CP (Andrews)
0.25	10.4	88.2	10.3458	90.9766	10.68	87.8
0.5	10.7	87.8	10.5771	89.5984	10.97	86.97
0.75	10.7	86.5	10.8122	88.1762	11.26	86.11
1	10.8	85.6	11.0513	86.7176	11.57	85.22
1.25	10.9	83.7	11.2947	85.2279	11.89	84.31
1.5	11.2	82.3	11.5428	83.7106	12.21	83.36
1.75	11.5	81	11.7960	82.1679	12.55	82.39
2	11.8	80.4	12.0546	80.6010	12.9	81.38
2.33	12.2	79.2	12.4048	78.4971	13.38	80
2.58	12.6	78.7	12.6775	76.8768	13.75	78.91
2.75	12.8	77.7	12.8668	75.7621	14.02	78.13
3	13	76	13.1511	74.1040	14.42	76.99
3.25	13.4	74.9	13.4431	72.4236	14.83	75.79
3.5	13.7	73	13.7431	70.7209	15.27	74.54
3.75	13.9	71.5	14.0517	68.9959	15.71	73.25
4	14.4	69.5	14.3695	67.2487	16.18	71.91

Table 5.5: Comparison for K-13 (Continued)

4.25	14.9	67.9	14.6971	65.4794	16.66	70.52
4.5	15.2	65.8	15.0350	63.6877	17.16	69.07
4.75	15.6	63.8	15.3840	61.8739	17.68	67.56
5	16.1	62	15.7446	60.0379	18.23	65.99
5.25	16.4	60.2	16.1177	58.1797	18.79	64.35
5.5	16.7	58	16.5039	56.2993	19.38	62.65
5.75	17.2	55.9	16.9040	54.3967	20	60.86
6	17.8	53.9	17.3188	52.4718	20.64	59.03
6.25	18	52	17.7493	50.5248	21.31	57.08
6.5	18.3	49.9	18.1964	48.5556	22.02	55.04
6.75	18.9	47.7	18.6609	46.5642	22.76	52.9
7	19.4	45.6	19.1439	44.5505	23.54	50.65
7.25	20	43.7	19.6465	42.5147	24.35	48.29
7.5	20.5	41.7	20.1697	40.4567	25.22	45.79
7.75	20.8	39.2	20.7148	38.3765	26.13	43.16
8	21.3	37.8	21.2829	36.2740	27.09	40.38
8.25	21.9	35	21.8753	34.1494	28.11	37.43

Table 5.5: Comparison for K-13 (Continued)

8.5	22.1	32.1	22.4935	32.0026	29.2	34.3
8.75	22.6	31.1	23.1387	29.8336	30.35	30.96
9	23.2	29.7	23.8126	27.6423	31.58	27.4
9.25	23.9	27.2	24.5167	25.4289	32.88	23.65
9.5	24.6	25.4	25.2526	23.1933	34.29	19.58
9.75	25.4	23.5	26.0220	20.9354	35.79	15.25
10	26.2	19.1	26.8268	18.6554	37.37	10.68
10.25	27.1	18.4	27.6690	16.3532	38.95	6.123
10.5	27.9	15.6	28.5504	14.0288	40.33	2.12
10.75	29	12.6	29.4732	11.6821	40.98	0.2543
11	30.1	10.6	30.4397	9.3133	41.06	0.01485
11.25	31.4	7.25	31.4522	6.9223	41.07	0.0008172
11.5	32.5	4.82	32.5130	4.5090	41.07	0.00004304
11.75	33.9	2.66	33.6249	2.0736	41.07	0.000002266
12	34.7	0.81	34.7904	0.8840	41.07	1.193E-07
12.25	35.5	0	36.0125	-0.2864	41.07	6.281E-09

CHAPTER 6

SUMMARY AND CONCLUSIONS

In the previous chapters, in the context of a discussion of limitations of traditional methods and an history overview of the development of artificial intelligence, EPNN was introduced and its usefulness was demonstrated in the modeling of three typical chemical processes.

More precisely, Chapter 1 introduced the history of chemical process modeling and the necessity for developing data based modeling approaches. In Chapter 2 some of the limitations of traditional thermodynamic modeling methods found in the literature were discussed. A brief historical overview of artificial intelligence was also included in Chapter 2, followed by a more focused discussion of artificial neural networks and evolutionary computing techniques and their limitations. Then the current trends in the development of artificial intelligence were introduced and the need for developing a new hybrid method in process modeling was discussed.

In Chapter 3, the objective of the dissertation was presented as a guideline. Following the dissertation objective, Chapter 4 included a detailed discussion of the EPNN modeling system: its structures, compositions, algorithms, modeling parameters and their considerations as well as EPNN implementation.

Applications in three distinctive chemical processes were demonstrated in Chapter 5. Comparisons with traditional neural networks were made in the dynamic neutralization process and the aqueous two phase system. Model reduction

of a biodegradation process was presented as the third application, showing the advantage of using EPNN over the traditional Andrew equation based model.

From the discussed example cases, it is clear that the EPNN approach is capable of modeling complex, dynamic or steady-state systems. EPNN can be also used to generate empirical symbolic formulae for chemical processes where determination of traditional modeling parameters is extremely difficult or even non-applicable.

In summary, the EPNN modeling system has the following features:

- Feedback links in EPNN network can be formed through training(evolution) to perform multi-step ahead prediction for dynamic nonlinear systems.
- Unlike those applications combining neural networks and genetic algorithms, symbolic formulae can be extracted from EPNN modeling results for further theoretical analysis and process optimization.
- EPNN system can also be used for data prediction tuning. In which case, only a minimum number of initial system conditions need to be adjusted. Therefore, the network structure of EPNN is more flexible and adaptable than traditional neural networks.
- Due to the polymorphic and evolutionary nature of the EPNN system, the initially randomized values of constants in EPNN networks will converge to the same or similar forms of functions in separate runs until the training process ends. The EPNN system is not sensitive to differences in initial values of the EPNN population. However, if there exists significant larger noise in one or

more data sets in the whole data composition, the EPNN system will probably fail to converge to a satisfactory level of prediction on these data sets.

- EPNN networks with a relatively small number of neurons can achieve similar or better performance than both traditional thermodynamic and neural network models.

The developed EPNN approach provides alternative methods for efficiently modeling complex, dynamic or steady-state chemical processes. EPNN is capable of producing symbolic empirical formulae for chemical processes, regardless of whether or not traditional thermodynamic models are available or can be applied. The EPNN approach does overcome some of the limitations of traditional thermodynamic/transport models and traditional neural network models.

CHAPTER 7

LIMITATIONS AND FUTURE WORK

Although the developed EPNN approach has many promising advantages over existing thermodynamic or transport models or even traditional artificial intelligence techniques, it is still important to point out the known issues and limitations of current EPNN modeling techniques.

Because EPNN is based on genetic programming, due to the stochastic nature of evolutionary computing, EPNN is not guaranteed to converge in all cases. When a large number of nodes (i.e. over 20) are used on the testing platform (Pentium II 233MHz, 96MB RAM and Windows NT 4.0), the EPNN evolves very slowly. This may be caused by a large search space or too many possible solutions for the system. For systems with a small number of inputs/outputs variables (less than 20), or for systems that can be reduced into subsystems of such scale, EPNN is suitable and is an efficient way to establish empirical models.

In other cases, when strong or radical interactions are coexisting with the inputs, EPNN may fail to converge or yield any meaningful results at all. This occurred in one of our pre-screening experiments, which involved the prediction of organic waste biodegradation rate constants based on the group contribution method (Tabak and Govind, 1993).

It may also be necessary to develop an improved version of the CME function (Equation 4.6) when many simultaneous outputs are desired. One of the possible improvements is designing CME as a n-dimensional weighted vector containing n

different mean square errors as shown in Equation 7.1. Then in the n-dimensional vector space, the objective of evolution will be minimizing the CME vector to obtain maximum fitness.

$$\overrightarrow{\boldsymbol{CME}} = (w_1 \times R_1, w_2 \times R_2, w_3 \times R_3, \ldots, w_n \times R_n) \tag{7.1}$$

where R_i is defined by Equation 4.7 and w_i is its associated weight constant.

As a performance limitation, EPNN can evolve or converge very slowly in some highly nonlinear or noisy environments. For example, in the case of biodegradation model reduction, it took the EPNN system more than 6,000 generations before it reached a satisfactory solution. Though the whole process was automatically controlled, it still took several hours (4 to 6 hours) in the testing platform. This may be due to the highly nonlinear relationship between the biodegradation rate and the initial concentrations in real world as compared to the simple Andrew model.

It is necessary to develop a more comprehensive diversity control over the gene pool when given a large search space. A search-space-size calculation table can be integrated into the EPNN evolutionary strategies. The table can be used as a validation tool for random formulae evolved from EPNN. In order to optimize EPNN modeling processes, the size of the search space table may be significantly reduced by incorporating a high-level of existing chemistry or physical knowledge of the target process. In such cases, more constraints can be added into the validation method of the search space table. The search space table can be also used to create formula trees of uniformly distributed depth. However, too strict constraints may yield non-converging models. Therefore, it is necessary to develop adaptive constraints in large

search spaces. Some further discussions can be found elsewhere (Chakraborty, 1999).

In more radical environments, co-evolution of interactive species may be more desirable than single species evolution in the current EPNN system. Some recent studies on co-evolution can be found in the literature (Puppala et al., 1998; Hirasawa et al., 2000; Berlanga et al., 2000). During co-evolution, one set of algorithms can be used for modeling the detailed chemical engineering process, while another set of algorithms can be used to monitor, evolve and modify the algorithm parameters or constants. In this way, more directed or self-adjustable evolving meta-algorithms can be developed.

Furthermore, it may be an effective way to include fitness distributions to design more efficient and faster converging systems (Fogel and Ghozeil, 1996). Parallel computing techniques may also be worthy of serious consideration to improve the computing efficiency and develop a more computing-powerful EPNN system.

In some engineering cases, if strong noise coexists in the process inputs, or the process is conducted in a noisy environment, it is necessary to develop methods to consider the noise in the model and can still properly model the process. Such improvement efforts can be based on noise processing models from recent developments (Beyer, 2000; Wong et al., 2000).

APPENDIX A

SOURCE CODES

The C ++ source code for EPNN modeling system.

```
#include <iostream>
#include <vector>
#include <list>
#include <map>
#include <string>
#include <algorithm>
#include <fstream>
#include <sstream>
#include <math.h>
#include <string.h>

extern "C"
{
#include<time.h>
}


#define EPNN_VER 2.1

using namespace std;


//*******************************************
//*  Class Parameters
//*******************************************

class Parameters {
public:
        Parameters();

        bool allowFeedback;
        int Num_Of_Inputs;
        int numNodes;
        int Num_Of_Outputs;
        double deltaData;
        int steps;  //how many steps calc for each individual

        int maxDepth;

        double qDataTypeChange;
        double qNodeType;
        double qNodeTypeChange;
        double qNodeFuncChange;

        int gpoolsize;
        int mpoolsize;
        double mutate_rate;
        int docking;
        double dif_eps;
        bool isSigDiffer(double, double);
```

```
        double desired_fitness;
        double mild_diversity;
        double min_diversity;
        double min_qReplacement;
        void diversity_control(double);
        double max_eps;
        double max_output;

        //function pack
        double Operate(int,double, double);
        bool isSingleFunc(int);
        int numFunctions;
        string fWriteOut(int, string s1, string s2);

        void write_diverse(ostream & os);

        void WriteOut(ostream & ) const;
        void ReadIn(string s);

private:
        double mr,qdtc, qnfc, qntc;
        void save_diverse(void);
        void restore_diverse(void);

};

void Parameters::ReadIn(string s)
{
        ifstream is(s.c_str());

        if(!is)
                throw "Parameter File Not Found!";

        is>>allowFeedback;
        is>>Num_Of_Inputs;
        is>>Num_Of_Outputs;
        is>>numNodes;
        is>>gpoolsize;
        is>>mpoolsize;
        is>>desired_fitness;
        is>>mild_diversity;
        is>>min_diversity;
        is>>min_qReplacement;


}

void Parameters::WriteOut(ostream & os) const
{

        os<<allowFeedback<<" ";
        os<<Num_Of_Inputs<<" ";
        os<<Num_Of_Outputs<<" ";
        os<<numNodes<<" ";
        os<<gpoolsize<<" ";
        os<<mpoolsize<<" ";
        os<<desired_fitness<<" ";
        os<<mild_diversity<<" ";
        os<<min_diversity<<" ";
        os<<min_qReplacement<<" ";

}
```

```
void Parameters::write_diverse(ostream & os)
{
        os<<endl<<"Mutation Parameters"<<endl;
        os<<mutate_rate<<" "<<qDataTypeChange<<" ";
        os<<qNodeTypeChange<<" "<<qNodeFuncChange<<endl;
}

void Parameters::save_diverse(void)
{
        mr=mutate_rate;
        qdtc=qDataTypeChange;
        qnfc=qNodeFuncChange;
        qntc=qNodeTypeChange;
}

void Parameters::restore_diverse(void)
{
        mutate_rate=mr;
        qDataTypeChange=qdtc;
        qNodeFuncChange=qnfc;
        qNodeTypeChange=qntc;
}

void Parameters::diversity_control(double div)
{
        if(div < mild_diversity ) {
                mutate_rate *=div;
                qDataTypeChange *=div;
                qNodeFuncChange *=div;
                qNodeTypeChange *=div;

                if(mutate_rate<0.001 ||
                        qDataTypeChange <0.001 ||
                        qNodeFuncChange <0.001 ||
                        qNodeTypeChange <0.001)

                        restore_diverse();

                return;
        }

        restore_diverse();

        return;
}

// determine whether or not is significantly different!!!

bool Parameters::isSigDiffer(double d1, double d2)
{
        if( d1==0 && d2 ==0) return false;

        if(d1==0 && d2 !=0) return true;
        if( d2==0 && d1 !=0) return true;

        if(d1/d2 > 1.5 || d2/d1 > 1.5)
                return true;

        if( fabs((d1-d2)/d2) > dif_eps) return true;

        return false;
```

```
}

string Parameters::fWriteOut(int idx, string s1, string s2)
{
        string res;
        if(idx>=numFunctions || idx <0 ) return res;

        switch (idx)
        {
        case 0:
                res=s1;
                res += "+";
                res +=s2;
                break;
        case 1:
                res=s1;
                res += "-";
                res +=s2;
                break;
        case 2:
                res=s1;
                res += "*";
                res +=s2;
                break;
        case 3:
                res=s1;
                res += "/";
                res +=s2;
                break;
        case 4:
                res="exp(";
                res+=s2;
                res+=")";
                break;
        default:
                break;
        }

        return res;
}

bool Parameters::isSingleFunc(int idx)
{
        bool yes=false;

        // if(idx==5) yes=true;

        return yes;

}

double Parameters::Operate(int idx, double d1, double d2)
{
        double res=0;

        switch(idx)
        {
        case 0:
                res=d1+d2;
                break;
        case 1:
                res=d1-d2;
```

```
                break;
        case 2:
                res=d1*d2;
                break;
        case 3:
                if(d2==0) res=0;
                else res=d1/d2;
                break;
        case 4:
                res=exp(d2);
                break;
        default:
                break;
        }

        return res;
}

Parameters::Parameters() {
        allowFeedback=false;
        Num_Of_Inputs=1;
        numNodes=1;
        Num_Of_Outputs=1;
        deltaData=0.1;

        numFunctions=4;
        //maybe changed accordingly

        maxDepth=120;

        qNodeType=0.65;
        qDataTypeChange=0.6;
        qNodeFuncChange=0.6;
        qNodeTypeChange=0.65;
        steps=3;

        gpoolsize=100;
        mpoolsize=50;
        mutate_rate=0.5;
        docking=2;
        dif_eps=0.005;
        desired_fitness=500;

        max_eps=60000.0;
        max_output=60000.0;

        mild_diversity=0.18;
           //threshold for parameter diversity control
        min_diversity=0.18;
                //threshold for pool diversity control
        min_qReplacement=0.2;
           //threshold for minimum replacement rate.

        save_diverse();
}

Parameters paras;

//*********************************************
//*  Class RandomData to Generate Random Data
//*********************************************
```

```
class RandomData {

public:
        double GenRanData(void);
        int GenRanInt(int);
        RandomData() ;

private:
        const unsigned long int Modulus;
        unsigned long int seed;
        unsigned long int result;

};

RandomData::RandomData() : Modulus(2147483647)
{
        double temp;
        seed=result=0;

        for(int i=0; i<200; i++)
                temp=GenRanData();

        return;
}

double RandomData::GenRanData(void)
{
        seed=(unsigned long int ) (seed +
                (unsigned long int) time(NULL)*31);
        seed= (unsigned long int ) ( seed ) % Modulus;

        if ( result==0 ) result=seed;

        result=result+seed;

        result=(unsigned long int ) result * 31 % Modulus;


        return (unsigned) (result-0.5) / (double) (Modulus);

}

//Generate random integer between 0 and d-1

int RandomData::GenRanInt(int d)
{
        double temp=GenRanData();
        int k=(int) (temp*d);
        if (k>=d) k=0;
        return k;
}

RandomData rans;

//****************************************************
//*  class DataPool to hold input/node values
//****************************************************
class DataPool {
public:
        double *  inputs;
```

```
        vector<double> inPool;
        double * nodes;

        int dpSize;
        int numDP;

        int curNode;
        bool isNodeInit;

        DataPool();
        ~DataPool();
        void stepInput(int);
        void regulize(void);

};


void DataPool::regulize(void)
{


        for(int i=0; i<paras.numNodes; i++)
        {
                if(nodes[i] > paras.max_output)
                        nodes[i]=paras.max_output;

        if(nodes[i] < (-1.0)*paras.max_output)
                        nodes[i]= (-1.0)* paras.max_output;




      /*
          // not allow negative values !
                if(nodes[i]< -0.01)
                        nodes[i]=paras.max_output;
                if(nodes[i]<0)
                        nodes[i]=0.0;

                */

        }
}

void DataPool::stepInput(int curDP)
{

        for(int i=0; i<paras.Num_Of_Inputs; i++)
        {
                inputs[i]=inPool[curDP*dpSize+i];
        }

}


DataPool::~DataPool()
{
        delete [] inputs;
        delete [] nodes;

}
```

```
DataPool::DataPool() {

        inputs=new double [paras.Num_Of_Inputs];
        for(int i=0; i<paras.Num_Of_Inputs; i++)
                inputs[i]=0;

        nodes=new double [paras.numNodes];
        for(i=0; i<paras.numNodes; i++)
                nodes[i]=0;

        curNode=0;
        isNodeInit=true;
        inPool.clear();

}

DataPool dpool;

//*************************************************
//*  class ElementalData to hold number, input or
//*        feedbacks
//*************************************************

class ElementalData {

public:
        ElementalData();
        void Mutate(void);
        double value(void);

        void WriteOut (ostream &) const;
        void ReadIn (istream &);

        friend ostream & operator<<(ostream &, const ElementalData &);

        bool isConst(void) { return type==0; }

        ElementalData & operator= (double);

private:
        double constant;
        int inindex;
        int findex;
        int type;
        void ReGenerate(void);
};

ElementalData & ElementalData::operator = (double x)
{
        type=0;
        inindex=-1;
        findex=-1;
        constant=x;

        return (*this);
}

ostream & operator<< (ostream & os, const ElementalData & ed)
{
        switch(ed.type)
```

```
        {
        case 0:
                os<<ed.constant;
                break;
        case 1:
                os<<'d'<<ed.inindex;
                break;
        case 2:
                os<<'N'<<ed.findex;
                break;
        default:
                break;
        }

        return os;
}

void ElementalData::WriteOut(ostream & os) const
{
        os<<type<<' '
          <<constant<<' '
          <<inindex<<' '
          <<findex<<' ';

        return;
}

void ElementalData::ReadIn(istream & is)
{

        is>>type;
        is>>constant;
        is>>inindex;
        is>>findex;
        return;
}

double ElementalData::value(void)
{

        switch (type) {
        case 0:
                return constant;
        case 1:
                return dpool.inputs[inindex];
        case 2:
                return dpool.nodes[findex];
        default:
                break;
        }

        return constant;
}

void ElementalData::Mutate(void)
{
        double mtype=rans.GenRanData();

        if(mtype > paras.qDataTypeChange)
        {
                ReGenerate();
                return;
        }
```

```
        switch (type)
        {
        case 0:
                if(rans.GenRanData() < 0.5)
                        constant +=rans.GenRanData() * paras.deltaData;
                else
                        constant -=rans.GenRanData() * paras.deltaData;
                 break;

        case 1:
                inindex= rans.GenRanInt(paras.Num_Of_Inputs);
                break;

        case 2:

                findex=rans.GenRanInt(paras.numNodes);

                if(! paras.allowFeedback)
                        while(findex < paras.Num_Of_Outputs)
                                findex=rans.GenRanInt(paras.numNodes);
                break;

        default:
                break;
        }

}


ElementalData::ElementalData()
{
        type=0;
        constant=0;
        findex=0;
        inindex=0;
        ReGenerate();
}

void ElementalData::ReGenerate ()
{

        type=rans.GenRanInt(3);

        if(paras.numNodes == paras.Num_Of_Outputs)
                type=rans.GenRanInt(2);

        if(! paras.allowFeedback  &&
                dpool.isNodeInit && type==2)
                type=1;


        switch (type) {

        case 0:
                constant=rans.GenRanData();
                break;

        case 1:
                inindex= rans.GenRanInt(paras.Num_Of_Inputs);
                break;
```

```
        case 2:

                findex=rans.GenRanInt(paras.numNodes);

                if(! paras.allowFeedback)
                        while(findex < paras.Num_Of_Outputs)
                                findex=rans.GenRanInt(paras.numNodes);
                break;

        default:
                break;

        }

}

//*************************************************
//*  Class TreeNode: holds the function tree
//*************************************************

int depth=0;

class TreeNode {
public:
        double value(void);
        void Mutate(void);
        TreeNode();
        TreeNode(int);
        TreeNode(char);  //define zero ground tree;
        ~TreeNode();
        bool isSingle(void);

        friend ostream & operator<<(ostream &, const TreeNode &);
        void WriteOut(ostream & os) const;
        void ReadIn(istream & is);
        TreeNode & operator=(const TreeNode &);

        void clear(void);
private:
        int type;
        int funindex;
        ElementalData data;
        TreeNode * lchild, * rchild;
        void funcMutate(void);

};

TreeNode & TreeNode::operator = (const TreeNode & tr)
{
        clear();

        type=tr.type;
        funindex=tr.funindex;
        data=tr.data ;

        if(type==0)
          return (*this);

        if(tr.lchild) {
                lchild=new TreeNode(0);
                (* lchild) = (* tr.lchild );
```

```
        }

        if(tr.rchild) {
                rchild=new TreeNode(0);
                (* rchild) = (* tr.rchild );
        }

        return (*this);

}

void TreeNode::clear(void)
{
        if(lchild) delete lchild;
        if(rchild) delete rchild;

        lchild=rchild=0;

        type=0;
        funindex=-1;
}

bool TreeNode::isSingle(void)
{
        if(type==0) return true;
        if(paras.isSingleFunc(funindex)) return true;

        return false;
}

void TreeNode::WriteOut(ostream & os) const
{
        os<<type<<' ';
        data.WriteOut(os);
        os<<funindex<<' ';

        if(lchild)
                lchild->WriteOut(os);
        if(rchild)
                rchild->WriteOut(os);

        if(type !=0)
                os<<' ';
}

void TreeNode::ReadIn(istream & is)
{
        clear();

        is>>type;
        data.ReadIn(is);
        is>>funindex;

        if(type==0)
        {
                if(lchild) delete lchild;
                if(rchild) delete rchild;
                lchild=rchild=0;

                return;
        }
```

```
        if(lchild) delete lchild;
        if(rchild) delete rchild;
        lchild=rchild=0;


        lchild=new TreeNode(1);
        lchild->ReadIn(is);

        if(! paras.isSingleFunc(funindex))
        {
                rchild=new TreeNode(1);
                rchild->ReadIn(is);
        }

        return;

}

ostream & operator<<(ostream & os, const TreeNode & tr)
{
        if(tr.type==0) {

                os<<tr.data;
                return os;
        }

        if(paras.isSingleFunc(tr.funindex)) {
                string s1, s2;

                ostringstream oss;
                oss<<(* tr.lchild);

                if(! (tr.lchild->isSingle()))
                        s1="(";
                s1 +=oss.str();

                if(! (tr.lchild->isSingle()))
                        s1 +=")";

                os<<paras.fWriteOut(tr.funindex,s1,s2);

                return os;

        }


           string s1, s2;

        ostringstream oss1, oss2;
        oss1<<(* tr.lchild);
        oss2<<(* tr.rchild);

        if(! (tr.lchild->isSingle()))
                s1="(";
        if(! (tr.rchild->isSingle()))
                s2="(";

        s1 +=oss1.str();

        if(! (tr.lchild->isSingle()))
                s1 +=")";
```

```
        s2 +=oss2.str();

        if(! (tr.rchild->isSingle()))
                s2 +=")";

        os<<paras.fWriteOut(tr.funindex,s1,s2);

        return os;
}

TreeNode::~TreeNode()
{
        if(lchild) delete lchild;
        if(rchild) delete rchild;

        lchild=rchild=0;

}

TreeNode::TreeNode(int c)
{
        lchild=rchild=0;
        funindex=-1;
        type=0;
}


TreeNode::TreeNode()
{

        lchild=rchild=0;
        funindex=-1;
        type=0;

        if(rans.GenRanData() > paras.qNodeType)
                type=1;
        else type=0;

        if(depth > paras.maxDepth) type=0;


        switch(type)
        {
        case 0:
                break;
        case 1:
                funindex=rans.GenRanInt(paras.numFunctions);
                depth ++;
                lchild=new TreeNode;
                if(!paras.isSingleFunc(funindex))
                    rchild=new TreeNode;
                break;
        default:
                break;
        }

}


//define zero ground tree
```

```
TreeNode::TreeNode(char)
{

        lchild=rchild=0;
        funindex=-1;
        type=0;
        depth=0;    //zero ground depth !

        if(rans.GenRanData() > paras.qNodeType)
                type=1;
        else type=0;

        if(depth > paras.maxDepth) type=0;

        switch(type)
        {
        case 0:
                break;
        case 1:
                funindex=rans.GenRanInt(paras.numFunctions);
                depth ++;
                lchild=new TreeNode;
                if(!paras.isSingleFunc(funindex))
                    rchild=new TreeNode;
                break;
        default:
                break;
        }

}

double TreeNode::value(void)
{
        double res;

        if(type==0)
                res=data.value();
        else
           {

                if(lchild ==0)
                        throw "Empty tree evaluation!";

                if(paras.isSingleFunc(funindex))
                        res=paras.Operate(funindex,lchild->value(),0);

                else
                {
                        if(rchild==0)
                                throw "Empty tree evaluation!";

                        res=paras.Operate(funindex,
                                lchild->value(), rchild->value());
                }
        }


        return res;

}

void TreeNode::Mutate(void)
```

```
{
        bool change=rans.GenRanData() > paras.qNodeTypeChange;

        if(!change)
        {
                switch(type){
                case 0:
                        data.Mutate();
                        break;
                case 1:
                        funcMutate();
                        break;
                default:
                        break;

                }

                return;
        }


        //type change mutation!

        if(type==0) {
                type=1;
                depth=0; // reset depth for adding a new subtree;

                funindex=rans.GenRanInt(paras.numFunctions);
                depth ++;
                lchild=new TreeNode;
                if(!paras.isSingleFunc(funindex))
                    rchild=new TreeNode;

                return;
        }

        if(type==1) {
                type=0;

                depth=0; //reset depth for subtree;

                if(lchild) delete lchild;
                if(rchild) delete rchild;

                lchild=rchild=0;

                return;
        }

        return;
}

void TreeNode::funcMutate(void)
{
        bool change=rans.GenRanData() > paras.qNodeFuncChange;

        // determine whether or not should change function types

        if(! change)
        {
                if(lchild) lchild->Mutate();
                if(rchild) rchild->Mutate();
```

```
                return;
        }


        depth=0;   //reset depth for changing functions;

        funindex=rans.GenRanInt(paras.numFunctions);

        if(paras.isSingleFunc(funindex) && rchild)
        {
                delete rchild;
                rchild=0;
                return;
        }

        if(! paras.isSingleFunc(funindex)  && !rchild)
        {
                depth ++;
            rchild=new TreeNode;
                return;
        }

        return;

}

//****************************************************
//*  class EPNNet defines a single EPNN individual net
//****************************************************
class EPNNet {
public:
        TreeNode * nodes;
        EPNNet();
        EPNNet(string s);  //read data from input;
        ~ EPNNet();
        double fitness(void);
        void invalidate(void) { fits=-1; curDP=0;}
        bool isValid(void);   // invalid net, should be discared !!!

        void Mutate(void);
        void incLife(double);
        void setLife(double);
        void CrossOver(EPNNet &);

        void WriteOut(ostream & os) const;
        void ReadIn(istream & is);
        friend ostream & operator<< (ostream &, const EPNNet &);

        double getFitness(void) const
        { return fits*(1.0-life); }

        void refresh(bool);  //recalculate the fitness !

        bool operator <(const EPNNet &) const;
        EPNNet & operator =(const EPNNet &);

        double Predict(int, int);


private:
        void InitInputs(string s);
        void stepInput(void);
```

```
        void calc(void);
        void stepCalc(void);
        double ems(int);

        int curDP;
        double fits;
        double life;

};

double EPNNet::Predict(int idx, int n)
{
        if(idx <0 || idx>=dpool.numDP)
                throw "Prediction out of range!";

        if(n<0 || n>= paras.Num_Of_Outputs)
                throw "Prediction out of range!";

        double res;
        int cur=curDP;
        curDP=idx;


    stepInput();
        for(int i=0; i<paras.steps; i++)
                stepCalc();

        res=dpool.nodes[n];

        curDP=cur;

        return res;

}

void EPNNet::setLife(double d)
{
        life=d;
}

EPNNet & EPNNet::operator = (const EPNNet & net)
{
        curDP=net.curDP;
        fits=net.fits;
        life=net.life;

        for(int i=0; i<paras.numNodes; i++)
                nodes[i]=net.nodes[i];

        return (*this);
}

bool EPNNet::operator < (const EPNNet & net) const
{
        return (getFitness()) >= (net.getFitness());
}

void EPNNet::CrossOver(EPNNet & net)
{
        int idx1=rans.GenRanInt(paras.numNodes);
        int idx2=rans.GenRanInt(paras.numNodes);
        TreeNode tr;
```

```
        // invode the copy operators !!!
        tr=net.nodes[idx1];
        net.nodes[idx1]=nodes[idx2];
        nodes[idx2]=tr;
}

void EPNNet::incLife(double d)
{
        life +=d;
        if(life>=1.0)
                invalidate();
}

void EPNNet::Mutate(void)
{
        for(int i=0; i<paras.numNodes; i++)
                nodes[i].Mutate();
}

EPNNet::~EPNNet()
{
        if(nodes)
                delete [] nodes;
}

ostream & operator<<(ostream & os, const EPNNet & net)
{
        for(int i=0; i<paras.numNodes; i++)
        {
                os<<endl<<"N"<<i<<"= ";
                os<<net.nodes[i];
        }

        return os;
}

void EPNNet::ReadIn(istream & is)
{
        for(int i=0; i<paras.numNodes; i++)
                nodes[i].ReadIn(is);
}

void EPNNet::WriteOut(ostream & os) const
{
        for(int i=0; i<paras.numNodes; i++)
        {
                os<<"\r\n";
                nodes[i].WriteOut(os);
        }

}

bool EPNNet::isValid(void)
{
        if(fits>=0) return true;
        return false;
}


//recalculate fitness !!!
```

```
void EPNNet::refresh(bool debug)
{

  char c;

  curDP=0;   //recalculation !


  double * eps=new double[paras.Num_Of_Outputs];
  double * emax=new double[paras.Num_Of_Outputs];
  double temp;

  for(int i=0; i<paras.Num_Of_Outputs; i++)
  {eps[i]=0; emax[i]=0; }


  for(i=0; i<dpool.numDP; i++ ) {
          calc();

          for(int j=0; j<paras.Num_Of_Outputs; j++)
          {
                  if(eps[j]>paras.max_eps) {
                          eps[j]=paras.max_eps;
                  }

                  temp=ems(j)*ems(j);

                  //adding max error consideration
                  if(temp>emax[j]) emax[j]=temp;

                  eps[j] +=temp;
          }


  }

  for(i=0; i<paras.Num_Of_Outputs; i++)
          if(eps[i]>paras.max_eps)
                  eps[i]=paras.max_eps*paras.max_eps
                                     *dpool.numDP;


  for(i=0; i<paras.Num_Of_Outputs; i++)
          eps[i]=sqrt(eps[i]/dpool.numDP);

  // adding max error consideration;
  for(i=0; i<paras.Num_Of_Outputs; i++)
  {
          if(emax[i]>paras.max_eps) emax[i]=paras.max_eps;
          eps[i]=0.7*eps[i]+0.3*sqrt(emax[i]);
  }


  for(i=0; i<paras.Num_Of_Outputs; i++)
          if(eps[i]>paras.max_eps)
                  eps[i]=paras.max_eps;
  fits=1.0;

  for(i=0; i<paras.Num_Of_Outputs; i++)
          fits *=eps[i];
```

```
  fits=exp(log(fits)/(double)(paras.Num_Of_Outputs) );

  if(fits!=0) fits=1.0/fits;
  else fits=paras.desired_fitness;

  fits=fits*(1.0-life);

  delete[] eps;
  delete[] emax;


}

double EPNNet::fitness(void)
{
  if(dpool.inPool.size() < paras.Num_Of_Inputs + paras.Num_Of_Outputs )
              throw "EPNN Net not proporly initialized!";

  if(fits >=0 ) return getFitness();

  refresh(false);  //recalculate fitness!!!!

  return getFitness();
}


//idx should be 0 -- Num_Of_Outputs -1
double EPNNet::ems(int idx)
{
        double res;
        res=dpool.inPool[(curDP-1)*dpool.dpSize + paras.Num_Of_Inputs + idx];
        res -= dpool.nodes[idx];

        // return absolute output difference;
        return res;
}

void EPNNet::calc(void)
{
        if(dpool.inPool.size() < paras.Num_Of_Inputs + paras.Num_Of_Outputs )
                throw "EPNN Net not properly initialized!";

        stepInput();
        for(int i=0; i<paras.steps; i++)
                stepCalc();

        fits=-1.0;
}

void EPNNet::stepCalc(void)
{

        vector<double> d(paras.numNodes);

        for(int i=0; i<paras.numNodes; i++)
                                d[i]=nodes[i].value();


        for(i=0; i<paras.numNodes; i++)
        {
```

```
                dpool.nodes[i]=d[i];
        }

        dpool.regulize();
}

void EPNNet::stepInput(void)
{
        dpool.stepInput(curDP);

        curDP ++;
}

void EPNNet::InitInputs(string s)
//initialize the data pool:: inputs !
{
        if(dpool.inPool.size() !=0)
                return;

        ifstream ifs;

        ifs.open(s.c_str());

        if(! ifs)
                throw "data file not file!";


        //read-in all the data into buffer;

        double dp=0;

        ifs>>dp;
        while(! (!ifs) ) {
                dpool.inPool.push_back(dp);
                ifs>>dp;
        }

        curDP=0;

        //*******************************************************
        dpool.dpSize=paras.Num_Of_Inputs+ paras.Num_Of_Outputs;
        dpool.numDP=dpool.inPool.size()/dpool.dpSize;
        //*******************************************************

        fits=-1;
        life=0.0;

}


EPNNet::EPNNet()
{

        nodes=new TreeNode[paras.numNodes];

        // dpool.inPool.clear();
        curDP=0;
        fits=-1;
        life=0.0;

        if(dpool.inPool.size() ==0)
```

```
        {dpool.dpSize=0; dpool.numDP=0;}

}

EPNNet::EPNNet(string s)
{

        dpool.isNodeInit=true;
        nodes=new TreeNode [paras.numNodes] ;
        dpool.isNodeInit =false;

        curDP=0;
        fits=-1;
        life=0.0;

        InitInputs(s);
}

//****************************************************
//*  class EPNNMoldel defines EPNN modeling system
//****************************************************

void netRefresh (EPNNet & net)
{

   net.refresh(false);
        // refresh the fitness calculation !!!
}

// End of definition of function object :)

class EPNNModel {
public:
        EPNNModel(string s);

        void Evolve(void);
        bool Finished(void);

        void WriteOut(ostream & os) const;
        void ReadIn(istream & is);
        friend ostream & operator<<(ostream &, const EPNNModel &);
        void WriteFittest(ostream & os) const;
        void WriteOut(string s) const;
        void WriteFittest(string s) const;
        void ReadIn(string s);
        void Predict(string s);
        void train_results(void);


private:
        list<EPNNet> genepool;
        list<EPNNet> mutpool;

        EPNNModel();
        void regenerate(string s);
        void renew(void);

        void refresh(void);
        void mutate(void);
        EPNNet & m_at(int);
        double diversity(void) const;
```

```
        void pool_merge(void);
        void newlife(list<EPNNet> &);
        void dyna_adjust(void);

        //***************************
        //*  Debug informations
        //***************************
        unsigned long int generation;
        unsigned int replacement;
        string data_file;

};

void EPNNModel::Predict(string s)
{
        ofstream ofs(s.c_str());

        if(!ofs)
                throw "Cannot create file for prediction!";

        ofs<<endl<<"Prediction Results"<<endl;
        ofs<<"Format: (Experimental Data) "
                <<"(Prediction Data) ...."<<endl;

        list<EPNNet>::iterator it=genepool.begin();

        int idx=0;

        for(int i=0; i<dpool.numDP; i++)
        {
                ofs<<endl;
                for(int j=0; j<paras.Num_Of_Outputs; j++)
                {
                        idx=i*(dpool.dpSize);
                        idx+=paras.Num_Of_Inputs + j;

                        ofs<<dpool.inPool[idx]<<" ";
                        ofs<<it->Predict(i,j)<<" ";
                }
        }

        return;
}

bool EPNNModel::Finished(void)
{
        double fit;
        fit=genepool.begin()->getFitness();

        if(fit>=paras.desired_fitness)
                return true;

        return false;

}

//renew at most 3 fittest individual in the net!!!
void EPNNModel::renew(void)
{
        EPNNet * net1, * net2, * net3;
        net1=net2=net3=0;
```

```
        double vals; int count=0;
        list<EPNNet>::iterator it;

        it=genepool.begin();
        vals=it->getFitness();
        net1 = new EPNNet(data_file);
        (*net1)= (*it); count ++;

        while(count <=3 && it != genepool.end())
        {
                if(paras.isSigDiffer(vals, it->getFitness()))
                {
                        count ++;

                        if(count ==2 )
                        {
                                net2 = new EPNNet(data_file);
                                (*net2)=(*it);
                        }

                        if(count==3)
                        {
                                net3=new EPNNet(data_file);
                                (*net3) = (*it);
                        }

                }

                it++;
        }


        regenerate(data_file);
        it=genepool.begin();

        if(net1) {
                (*it)=(* net1);
                delete net1;
                it++;
        }

        if(net2) {
                (*it)=(* net2);
                delete net2;
                it++;
        }

        if(net3) {
                (*it)=(* net3);
                delete net3;
                it++;
        }

}

// dynamically adjust modeling parameters
// to better fit the target system
void EPNNModel::dyna_adjust(void)
{
        double div, qr;
```

```
        div=diversity();
        paras.diversity_control(div);

        //calculate replacement ratio
        qr=(double)(replacement)/(double)(paras.mpoolsize);

        if(div<paras.min_diversity ||
                qr < paras.min_qReplacement)
                renew();

        return;

}

// set life=0 for each individuals in the list
void EPNNModel::newlife(list<EPNNet> & lis)
{
        list<EPNNet>::iterator it;
        for(it=lis.begin(); it !=lis.end() ; it++)
                it->setLife(0);
}

void EPNNModel::ReadIn(string s)
{
        ifstream ifs(s.c_str());
        if(!ifs)
                return;

        ReadIn(ifs);
}

void EPNNModel::WriteFittest(string s) const
{
        ofstream ofs(s.c_str());
        if(!ofs)
                throw "Cannot create output file!";
        WriteFittest(ofs);
}

void EPNNModel::WriteOut(string s) const
{
        ofstream ofs(s.c_str());

        if(!ofs)
                throw "Cannot create output file!";

        WriteOut(ofs);
}

void EPNNModel::pool_merge(void)
{

        int dock=0;
        list<EPNNet>::iterator mit;
        list<EPNNet>::const_iterator nit;

        nit=mutpool.begin();

        while(nit != mutpool.end() )
        {
                mit=genepool.begin();
```

```
                dock=0;

                while(dock < paras.docking && mit !=genepool.end() )
                {
                        if((*mit).getFitness() < (*nit).getFitness()) {
                                (* mit) = (* nit);
                                replacement ++;
                                (*mit).refresh(false);
                                mit++; dock ++;
                                continue;
                        }

                        if((*mit).getFitness() == (*nit).getFitness()) {
                                dock ++; mit ++;
                                continue;
                        }

                        if((*mit).getFitness() > (*nit).getFitness())
                                dock=0;

                        mit ++;

                }

                nit ++;
        }

}


double EPNNModel::diversity(void) const
{
        double div=0, prev=0;
        int count=0;

        list<EPNNet>::const_iterator
                it=genepool.begin();

        prev=it->getFitness();
        count++;

        for(it=genepool.begin(); it!=genepool.end(); it++)
        {
                if(paras.isSigDiffer(prev,it->getFitness())) {
                        count++;
                        prev=it->getFitness();
                }
        }

        div=(double) (count) / (double)(paras.gpoolsize);

        return div;
}


/*
double EPNNModel::diversity(void) const
{
        double div=0, prev=0;

        list<EPNNet>::const_iterator
```

```
                it=genepool.begin();

        prev=it->getFitness();
        it++;

        for(; it!=genepool.end(); it++)
        {
                div +=fabs(prev - it->getFitness())/prev;
                prev=it->getFitness();
        }

        div= div/(genepool.size()-1);

        return div*10.0;
}
*/


//refresh pool fitness
void EPNNModel::refresh(void)
{
   for_each(genepool.begin(), genepool.end(), netRefresh);
   genepool.sort();
}

ostream & operator<<(ostream & os, const EPNNModel & model)
{
        int count=0;
        os<<endl<<"Generation="<<model.generation<<endl;
        os<<"Diversity="<<model.diversity()<<endl;

        list<EPNNet>::const_iterator it;

        os<<"Gene Pool =>"<<endl;
        for(it=model.genepool.begin(); it!=model.genepool.end(); it++)
        {
                os<<it->getFitness()<<"  ";
                count++;

                if(count >50) break;
        }

        os<<endl;

/*
        os<<"Mutation Pool =>"<<endl;
        for(it=model.mutpool.begin(); it!=model.mutpool.end(); it++)
                os<<it->getFitness()<<"  ";

        os<<endl;
*/

        os<<"Replacement="<<model.replacement<<endl;

        return os;
}


void EPNNModel::WriteFittest(ostream & os) const
{
        //*******************************************************
```

```
        //*  Output the extra information for fittest individual
        //*****************************************************
        char buf[20];
        _strdate(buf);

        os<<endl;
        os<<"************************************************"<<endl;
        os<<"*  Information Regarding The Fittest Individual  *"<<endl;
        os<<"*                                                *"<<endl;
        os<<"*    Date:  "<<buf<<"                             *"<<endl;
        os<<"************************************************"<<endl;

        os<<"Fitness="<< genepool.begin()->getFitness()
                <<endl;
        os<<"Generation="<<generation<<endl;

        os<<"Inputs:";
        for(int i=0; i< paras.Num_Of_Inputs; i++)
                os<<'d'<<i<<' ';
        os<<endl;

        os<<"Outputs:";
        for(i=0; i< paras.Num_Of_Outputs; i++)
                os<<'N'<<i<<' ';
        os<<endl;


        os<<(* genepool.begin());
        os<<endl;
        os<<"********** End of Report ************************";
        os<<endl;

}

void EPNNModel::WriteOut(ostream & os) const
{
        int count=0;
        os<<generation<<endl;

        list<EPNNet>::const_iterator it;

        for(it=genepool.begin(); it!=genepool.end(); it++)
        {
                it->WriteOut(os);
                count ++;
                if(count >50) break;
        }

        WriteFittest(os);

}

void EPNNModel::ReadIn(istream & is)
{
        int count=0;
        is>>generation;

        list<EPNNet>::iterator it;

        for(it=genepool.begin(); it!=genepool.end(); it++)
        {
                it->ReadIn(is);
```

```
                count++;
                if(count>50) break;
        }

        refresh();
}

void EPNNModel::Evolve(void)
{
        dyna_adjust();

        replacement=0;
        for_each(genepool.begin(), genepool.end(), netRefresh);
        genepool.sort();

        //********************************************
        //**  Generate the mutation pool
        //********************************************


        list<EPNNet>::iterator nit, mit;

        nit=genepool.begin();
        mit=mutpool.begin();

        for(int i=0; i< paras.mpoolsize; i++)
        {
                (* mit) = (*nit);
                mit++; nit++;
        }


        mutate();

        for_each(mutpool.begin(), mutpool.end(), netRefresh);
        mutpool.sort();
        genepool.sort();

        pool_merge();
        generation ++;

        nit=genepool.begin();
        nit->refresh(true);

        return;
}

void EPNNModel::mutate(void)
{
        list<EPNNet>::iterator nit=mutpool.begin();
        int idx1=0;

        for(int i=0; i< paras.mpoolsize; i++, idx1++)
        {
                if(rans.GenRanData()<paras.mutate_rate)
                {
                        (* nit).Mutate();
                }

                else
```

```
			{
				int idx2=rans.GenRanInt(paras.mpoolsize);
				if(idx2 != idx1)
					(* nit).CrossOver(m_at(idx2));
			}
		}

}

EPNNet & EPNNModel::m_at(int idx)
{
		list<EPNNet>::iterator mit=mutpool.begin();

		int i=idx;

		while(i>0) {
			mit ++;
			i --;
		}

		return (* mit);
}


EPNNModel::EPNNModel()
{
		genepool.clear();
		mutpool.clear();
		EPNNet * net;

		//only initialize gene pool !
		for(int i=0; i<paras.gpoolsize; i++)
		{
			net=new EPNNet;
			genepool.push_back(* net);
		}

		//only initialize mutation pool !
		for(i=0; i<paras.mpoolsize; i++)
		{
			net=new EPNNet;
			mutpool.push_back(* net);
		}

		generation=1;
		replacement=0;
		data_file="";

}

void EPNNModel::regenerate(string s)
{
		genepool.clear();

		EPNNet * net;

		//only initialize gene pool !
		for(int i=0; i<paras.gpoolsize; i++)
		{
			net=new EPNNet(s);
			genepool.push_back(* net);
```

```
        }

        return;

}

EPNNModel::EPNNModel(string s)
{
    genepool.clear();
        mutpool.clear();
        EPNNet * net;

        //only initialize gene pool !
        for(int i=0; i<paras.gpoolsize; i++)
        {
                net=new EPNNet(s);
                genepool.push_back(* net);
        }

        //only initialize mutation pool !
        for(i=0; i<paras.mpoolsize; i++)
        {
                net=new EPNNet(s);
                mutpool.push_back(* net);
        }

        refresh();
        generation=1;
        replacement=0;

        data_file=s;
}

//********************************************************
//*  Main fucntion driver
//********************************************************

void show_usuage(void)
{
        cout<<endl;
        cout<<"*****************************************"<<endl;
        cout<<"*   EPNN Modeling Command Parameters    *"<<endl;
        cout<<"*                                       *"<<endl;
        cout<<"*    Version "<<EPNN_VER
                <<"                        *"<<endl;
        cout<<"*****************************************"<<endl;

        cout<<"Usage 1:"<<endl;
        cout<<"********************************"<<endl;
        cout<<"epnn paras.dat"<<endl;
        cout<<"********************************"<<endl;
        cout<<"(Creating modeling parameters.)"<<endl;
        cout<<"where \"paras.dat\" is the output file name "<<endl;
        cout<<"  to save input modeling parameters."<<endl;

        cout<<endl;

        cout<<"Usage 2:"<<endl;
        cout<<"********************************"<<endl;
        cout<<"epnn paras.dat data.dat"<<endl;
        cout<<"********************************"<<endl;
```

```
        cout<<"(Run Modeling Training.)"<<endl;
        cout<<"where \"paras.dat\" is the parameter file; "<<endl;
        cout<<"  \"data.dat\" is the normalized data file."<<endl;

        cout<<endl;

        cout<<"Usage 3:"<<endl;
        cout<<"**********************************"<<endl;
        cout<<"epnn /p paras.dat data.dat"<<endl;
        cout<<"**********************************"<<endl;
        cout<<"(Run Modeling Prediction.)"<<endl;
        cout<<"where \"paras.dat\" is the parameter file; "<<endl;
        cout<<"  \"data.dat\" is the normalized data file."<<endl;


}

void init_paras(const char * pfile)
{
        ofstream ofs(pfile);
        if(! ofs)
                throw "Can not create output file";

        cout<<endl;
        cout<<"Please Input your parameters one by one,"
                <<" using -1 for default."<<endl;

        double d; int i;

        //*********************************
        cout<<endl<<"allowFeedback=";
        cin>>i;
        if(i !=-1) paras.allowFeedback=i;

        cout<<endl<<"Num_Of_Inputs=";
        cin>>i;
        if(i!= -1) paras.Num_Of_Inputs=i;

        cout<<endl<<"Num_Of_Outputs=";
        cin>>i;
        if(i!= -1) paras.Num_Of_Outputs=i;

        cout<<endl<<"numNodes=";
        cin>>i;
        if(i!= -1) paras.numNodes=i;

        cout<<endl<<"gpoolsize=";
        cin>>i;
        if(i!= -1) paras.gpoolsize=i;

        cout<<endl<<"mpoolsize=";
        cin>>i;
        if(i!= -1) paras.mpoolsize=i;

        cout<<endl<<"desired_fitness=";
        cin>>d;
        if(d!= -1) paras.desired_fitness=d;

        //*****************************************

        cout<<endl<<"mild_diversity=";
```

```
        cin>>d;
        if(d!= -1) paras.mild_diversity=d;

        cout<<endl<<"min_diversity=";
        cin>>d;
        if(d!= -1) paras.min_diversity=d;

        cout<<endl<<"min_qReplacement=";
        cin>>d;
        if(d!= -1) paras.min_qReplacement=d;

        paras.WriteOut(ofs);

        return;

}

void prediction(const char * file1, const char * file2)
{
        int i;
        string pars(file1), dats(file2);
        string save1("farm.dat");
        string save2("results.dat");

        cout<<"Initializing...... Please wait...."<<endl;
        paras.ReadIn(pars);

        EPNNModel model(dats);
        model.ReadIn(save1);

        for(i=0; i<4; i++)
                model.Evolve();

        cout<<endl;
        cout<<"Predicting...... Please wait...."<<endl;
        model.Predict(save2);
        cout<<"Prediction finished!"<<endl;
        cout<<endl;
        cout<<"Please check ";
        cout<<"\""<<save2.c_str()<<"\" for details.";
        cout<<endl;

        return;
}

int main(int argc, char * argv[])
{

        try{

                switch(argc)
                {
                case 1:
                        show_usuage();
                        return 0;

                case 2:
                        init_paras(argv[1]);
                        return 0;

                case 3:
                        break;
```

```
        case 4:

                if(strcmp(argv[1],"/p") !=0 &&
                        strcmp(argv[1],"/P") !=0)
                        throw "Wrong command parameters!";

                prediction(argv[2],argv[3]);
                return 0;

        default:
                throw "Wrong command parameters!";

        }


        string pars(argv[1]), dats(argv[2]);
        string save1("farm.dat");
        string save2("fittest.dat");
        string training("trains.dat");

        paras.ReadIn(pars);

        EPNNModel model(dats);

        model.ReadIn(save1);

        do
        {
                cout<<"numDP="<<dpool.numDP<<endl;
                cout<<model;
                if(model.Finished()) break;
                model.Evolve();
                model.Predict(training);
                model.WriteFittest(save2);
                model.WriteOut(save1);
        }
        while(! model.Finished()) ;

        cout<<endl;
        cout<<"****************************************"<<endl;
        cout<<"*    Model Fitting has finished        *"<<endl;
        cout<<"****************************************"<<endl;

        cout<<" The EPNNModel is saved in ";
        cout<<"\""<<save1.c_str()<<"\"."<<endl;
        cout<<" The fittest individual is shown";
        cout<<" in \""<<save2.c_str()<<"\"."<<endl;

}

//****************************************
//*  Exception dealing system
//****************************************

catch ( const char * msg)
{
        cout<<endl;
        cout<<msg;
        cout<<endl;
}
```

```
        catch (bad_alloc &)
        {
                cout<<endl;
                cout<<"MEMORY CANNOT BE ALLOCATED!";
                cout<<endl;
                return -1;
        }


        catch (...)
        {
                cout<<endl;
                cout<<"Unknown error occured!";

        }

  return 0;
}
```

APPENDIX B

INTERNET RESOURCES

The presented dissertation is largely based on branches of modern soft computing and artificial intelligence. Because of rapid development and progress in these fields, the best way to track the most recent progress is to look up and search on the internet. Therefore, a brief collection of internet resources is listed below. The listed internet resources can be used as a starting point to search for most updated information on artificial intelligence and soft computing.

- **The Genetic Computing Notebook.**

  http://www.geneticprogramming.com/

- **USENET newsgroup for evolutionary computing.**

  news:comp.ai.genetic

- **FAQ for comp.ai.genetic**

  http://alife.santafe.edu/ joke/encore/www/

- **USENET newsgroup for neural networks.**

  news:comp.ai.neural-nets

- **FAQ for comp.ai.neural-nets**

  ftp://ftp.sas.com/pub/neural/FAQ.html

- **The Hitch-Hiker's Guide to Evolutionary Computation.**

  http://alife.santafe.edu/ joke/encore/

- **World Online Conference on Soft Computing (WSC).**

  http://www.bioele.nuee.nagoya-u.ac.jp/WFSC/

- **IEEE Neural Network Council Home Page.**

  http://www.ewh.ieee.org/tc/nnc/

- **International Neural Network Society**

  http://cns-web.bu.edu/INNS/index.html

- **Japanese Neural Network Society (JNNS), (in Japanese)**

  http://jnns.inf.eng.tamagawa.ac.jp/English/index-e.html

REFERENCES

Abrams, D. and J. Prausnitz, Statistical thermodynamics of liquid mixtures: a new expression for the excess Gibbs energy of partly or completely miscible systems. *AIChE Journal*, 21(1): 116 (1975).

Aho, A. V. et al., *Data Structures and Algorithms*. Addison-Wesley (1987).

Albertsson, P.-A., *Partition of Cell Particles and Macromolecules*. Wiley-Interscience, second edition (1971).

Alvarez, E., et al., Design of a combined mixing rule for the prediction of vapor-liquid equilibria using neural networks. *Industrial and Engineering Chemistry Research*, 38: 1706–1711 (1999).

Aminzadeh, F. and M. Jamshidi, *Soft Computing*. Prentice Hall (1994).

Anderson, J., *Logistic Discrimination in Classification, Pattern Recognition and Reduction of Dimensionality*, 169–191. Elsevier Science Ltd., Amsterdam: North Holland (1982).

Angeline, P. J., et al., An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1): 54–65 (1994).

Bäck, T., *Evolutionary Algorithms in Theory and Practice : Evolution Strategies, Evolutionary programming, Genetic algorithms*. Oxford University Press (1996).

Bäck, T., et al., Evolutionary computation: Comments on the history and current state. *IEEE Transactions on Evolutionary Computation*, 1(1): 3–17 (1997).

Bailey, D. and D. Thompson, How to develop neural networks. *AI Expert*, 5(6): 38 (1990).

Banzhaf, W. et al., editors, *Genetic Programming : An Introduction On The Automatic Evolution Of Computer Programs And Its Applications*. Morgan Kaufmann Publishers (1998).

Berenson, M. L. and D. M. Levine, *Basic Business Statistics: Concepts and Applications*. Prentice Hall International (1983).

Berlanga, A., et al., General learning co-evolution method to generalize autonomous robot navigation behavior. In *Proceedings of the IEEE Conference on Evolutionary Computation*, 769–776 (2000).

Beyer, H.-G., Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. *Computer Methods in Applied Mechanics and Engineering*, 186(2): 239–267 (2000).

Bhaskar, V., et al., Multiobjective optimization of an industrial wiped-film pet reactor. *AIChE Journal*, 46(5): 1046–1058 (2000).

Blas, F. J. and L. F. Vega, Prediction of binary and ternary diagrams using the statistical associating fluid theory (saft) equation of state. *Industrial and Engineering Chemistry Research*, 37(2): 660–674 (1998).

Brasquet, C. and P. L. Cloirec, QSAR for organics adsorption onto activated carbon in water: What about the use of neural networks? *Water Research*, 33(17): 3603–3608 (1999).

Bury, K. V., *Statistical Models In Applied Science*. Wiley International (1975).

Cao, H., et al., The kinetic evolutionary modeling of complex systems of chemical reactions. *Computers and Chemistry*, 23: 143–151 (1999).

Carsky, M. and D. Do, Neural network modeling of adsorption of binary vapour mixtures. *Adsorption*, 5(3): 183–192 (1999).

Caudill, M. and C. Butler, *Understanding Neural Networks: Computer Explorations*, volume 1–2. MIT Press (1992).

Chakraborty, G., Genetic programming for a class of constrained optimization problems. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, I314–I319 (1999).

Chambers, L., editor, *Practical Handbook of Genetic Algorithms*. CRC Press (1995).

Chen, S.-H. and C.-C. Ni, Evolutionary artificial neural networks and genetic programming: A comparative study based on financial data. *Artificial Neural Networks and Genetic Algorithms* (1997).

Chiou, J.-P. and F.-S. Wang, A hybrid method of differential evolution with application to optimal control problems of a bioprocess system. In *Proceedings on Evolutionary Computation, The 1998 IEEE International Conference*, 627 (1998).

Doherty, S., et al., Experiment design considerations for non-linear system identification using neural networks. *Computers and Chemical Engineering*, 21(3): 327–346 (1997).

Dorffner, G., Neural networks for time series processing. *Neural Network World*, 6(4): 447–468 (1996).

Edwards, K., et al., Kinetic model reduction using genetic algorithms. *Computers and Chemical Engineering*, 22 (1998).

Fausett, L., *Fundamentals of Neural Networks:Architectures, Algorithms and Applications*. Prentice Hall (1994).

Flory, P. J. et al., Statistical thermodynamic of chain molecule liquids. i. an equation of state for normal and paraffin hydrocarbons. *Journal of American Chemistry Society*, 86: 3507 (1964).

Fogel, D., *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press (1995a).

Fogel, D., *Evolutionary Computation: The Fossil Record*. IEEE Press (1998).

Fogel, D. B., Comparison of evolutionary programming and genetic algorithms on selected constrained optimization problems. *Simulation*, 64(6) (1995b).

Fogel, D. B., Pratical advantages of evolutionary computation. *Applications of Soft Computing*, 14–22 (1997).

Fogel, D. B. and A. Ghozeil, Using fitness distributions to design more efficient evolutionary computations. In *Proceedings of the IEEE Conference on Evolutionary Computation*, 11–19 (1996).

Fogel, D. B., et al., Evolving neural networks. *Biological Cybernetics* (1990).

Fraga, E. and T. Matias, Synthesis and optimization of a nonideal distillation system using a parallel genetic algorithm. *Computers and Chemical Engineering*, 20: s79–s84 (1996).

Fredenslund, A., et al., Group-contribution estimation of activity coefficients in nonideal liquid mixtures. *AIChE Journal*, 21(6): 1086 (1975).

Fredenslund, A., et al., *Vapor-Liquid Equilibria Using UNIFAC : A Group Contribution Method*. Elsevier Scientific Pub. Co., New York (1977).

Gagné, F. and C. Balise, Predicting the toxicity of complex mixtures using artificial neural networks. *Chemosphere*, 35(6): 1343–1363 (1997).

Gallant, S., et al., Modeling of non-linear elution of proteins in ion-exchange chromatography. *Journal of Chromatography*, 702: 125–142 (1995).

Galushko, S., Calculation of retention and selectivity in reversed phase liquid chromatography. iv: Chromdream software for the selction of initial conditions and for simulating chromatographic behavior. *Journal of Chromatography*, 552: 91–102 (1991).

Galvan, I., et al., The use of neural networks for fitting complex kinetic data. *Computers and Chemical Engineering*, 20 (1996).

Gao, F., et al., Genetic algorithms and evolutionary programming hybrid strategy for structure and weight learning for multilayer feedforward neural networks. *Industrial and Engineering Chemistry Research*, 38: 4330–4336 (1999).

Gao, Y.-L., et al., Thermodynamics of ammonium sulfate-polyethylene glycol aqueous two-phase systems. Part 2. Correlation and prediction using extended UNIFAC equation. *Fluid Phase Equilibria*, 63(1–2): 173–182 (1991).

Gen, M. and R. Cheng, *Genetic Algorithms and Engineering Design*. John Wiley (1997).

Ghosh, P., et al., Integrated product engineering: A hybrid evolutionary framework. *Computers and Chemical Engineering*, 24: 685–691 (2000).

Goldberg, D., *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley (1989).

Greeff, D. and C. Aldrich, Empirical modeling of chemical process systems with evolutionary programming. *Computers and Chemical Engineering*, 22(7–8): 995–1005 (1998).

Gruau, F., Genetic synthesis of modular neural networks. In *Proceedings of The Fifth International Conference on Genetic Algorithms*, 318–325 (1993).

Gu, T., *Mathematical Modeling and Scale-Up of Liquid Chromatography*. Springer (1995).

Gustaffsson, A. et al., The nature of phase separation in aqueous two-phase systems. *Polymer*, 27(21): 1768 (1986).

Hancock, P. J. B., *Coding Strategies for Genetic Algorithms and Neural Nets*. Ph.D. thesis, University of Stirling, UK (1992).

Hansen, J. V. and R. D. Nelson, Neural networks and traditional time series methods: A synergistic combination in state economic forecasts. *IEEE Transactions on Neural Networks*, 8(4): 863–873 (1997).

Hinde, R. F. and D. J. Cooper, Pattern-based approach to excitation diagnostics for adaptive process control. *Chemical Engineering Science*, 49: 1403–1415 (1994).

Hinton, G., How neural networks learn from experience. *Scientific American* (1992).

Hirasawa, K., et al., Genetic symbiosis algorithm. In *Proceedings of the IEEE Conference on Evolutionary Computation*, 1377–1384 (2000).

Holland, J., *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI (1975).

Hornik, K., et al., Multi-layer feed-forward networks are universal approximators. *Neural Networks*, 3 (1989).

Huenupi, E., et al., Optimization and design considerations of two-phase continuous protein separation. *Journal of Chemical Technology and Biotechnology*, 74(3): 256–263 (1999).

Huo, Q. and C. Chan, Contextual vector quantization for speech recognition with discrete hidden Markov model. *Pattern Recognition*, 28: 513 (1995).

Jarke, M. and W. Marquardt, Design and evaluation of computer-aided process modelling tools. In *International Conference on Intelligence Systems in Process Engineering*, AIChE Symposium Series, 97–109 (1996).

Jonathan, W. L., Randomization tests: Statistics for experimenters. *Computer Methods and Programs in Biomedicine*, 35(1): 43–51 (1991).

Kan, P. and C.-J. Lee, A neural network model for prediction of phase equilibria in aqueous two-phase extraction. *Industrial and Engineering Chemistry Research*, 35: 2015–2023 (1996).

Kiguchi, K., et al., Identification of robot manipulators using neural networks and genetic programming. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, volume 4, 802–806 (1999).

Kinnear, K., *Advances in Genetic Programming.* MIT Press, Cambridge, MA (1994).

Klein, E. J. and S. L. Rivera, Optimization of ion-exchange protein separations using a vector quantizing neural network. *Biotechnology Progress*, 16: 506–512 (2000).

Koza, J. R., *Genetic Programming: On the Programming of Computers By Means of Natural Selection.* MIT Press (1992).

Koza, J. R., *Genetic Programming II : Automatic Discovery of Reusable Programs.* MIT Press (1994).

Krothapally, M. and S. Palanki, A neural network strategy for batch process optimization. *Computers and Chemical Engineering*, 21: S463–S468 (1997).

Kuscu, I. and C. Thornton, Design of artificial neural networks using genetic algorithms: review and prospect. Cognitive and computing sciences, University of Sussex (1994).

Lakshminarayanan, S., et al., New product design via analysis of historical databases. *Computers and Chemical Engineering*, 24(2): 671–676 (2000).

Lei, X., et al., Equilibrium phase behavior of the poly(ethylene glycol)/potassium phosphate/water two-phase system at 4 c. *J. Chem. Eng. Data*, 35: 420–423 (1990).

Lennox, B. et al., Case study of investigating the application of neural networks for process modeling and condition monitoring. *Computers and Chemical Engineering*, 22(11): 1573–1579 (1998).

Lentner, M., *Introduction To Applied Statistics*. Weber and Schmidt (1975).

Leung, Y., et al., Degree of population diversity-a perspective on premature convergence in genetic algorithms and its markov chain analysis. *IEEE Transactions on Neural Networks*, 8(5): 1165–1176 (1997).

Luo, L., et al., Adaptive quantization algorithm for MPEG-2 video coding. In *Proceeddings of IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 5, 2841–2844 (1998).

Luo, R. G. and J. T. Hsu, Optimization of gradient profiles in ion-exchange chromatography for protein purification. *Industrial and Engineering Chemistry Research*, 36(2): 444–450 (1997).

Magnussen, T., et al., UNIFAC parameter table for prediction of liquid-liquid equilibria. *Industrial and Engineering Chemistry, Process Design and Development*, 20(2): 331–339 (1981).

Mandal, D. K., *Modelling of In-Situ Bioremediation With Emphasis on Inhibitory Kinetics and Biomass Growth*. Ph.D. Thesis, New Jersey Institute of Technology, Newark, New Jersey (1998).

Mao, Q. M. and M. Hearn, Optimization of affinity and ion-exchange chromatographic processes for the purification of proteins. *Biotechnology and Bioengineering*, 52(2): 204–222 (1996).

Maren, A., et al., *Handbook of Neural Computing Applications*. Academic Press (1990).

Masters, T., *Advanced Algorithms for Neural Networks: A C++ Sourcebook*. John Wiley and Sons (1995).

Mavrovouniotis, M. L., *Artificial Intelligence in Process Engineering*. Academic Press (1990).

McAvoy, T. J., et al., Dynamics of pH in CSTRs. *Industrial Engineering and Chemistry Process Research*, 11: 68–70 (1972).

McCulloch, W. S. and W. Pitts, A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5: 115–133 (1943).

McKay, B., et al., Steady-state modeling of chemical process systems using genetic programming. *Computers and Chemical Engineering*, 21(9): 981–996 (1997).

Mendenhall, W. and R. L. Scheaffer, *Mathematical Statistics With Applications*. Duxbury Press (1973).

Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, second edition edition (1994).

Mitchell, M., *An Introduction to Genetic Algorithms*. MIT Press (1996).

Morari, M. and E. Zafiriou, *Robust Process Control*. Prentice Hall, Englewood Cliffs, NJ (1989).

Moriaty, D. E. and R. Miikkulainen, Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation*, 5 (1998).

Moros, R., et al., A genetic algorithm for generating initial parameter estimations for kinetic models of catalytic processes. *Computers and Chemical Engineering*, 20 (1996).

Myers, A. L., et al., Prediction of multicomponent adsorption equilibrium. discussions on energy of adsorption in relation to adsorption equilibrium. *International Chemical Engineering*, 32(3): 585–590 (1992).

Nelson, M. M. and W. Illingsworth, *A Practical Guide to Neural Networks*. Addison-Wesley (1990).

Nikravesh, M., et al., Model identification of nonlinear time variant processes via artificial neural network. *Computers and Chemical Engineering*, 20(11) (1996).

Oliveira, K. A., et al., Using artificial neural networks to forecast chaotic time series. *Physica A: Statistical Mechanics and Its Applications*, 284(1): 393–404 (2000).

Omatu, S., et al., *Neuro-control and Its Applications*. Springer–Verlag (1996).

Pham, Q., Dynamic optimization of chemical engineering processes by an evolutionary method. *Computers and Chemical Engineering*, 22: 1089–1097 (1998).

Press, W. H., et al., *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press (1993).

Puppala, N., et al., Shared memory based cooperative coevolution. In *Proceedings of the IEEE Conference on Evolutionary Computation*, 1132–1136 (1998).

Quinlan, P. T., Structural change and development in real and artificial neural networks. *Neural Networks*, 11(4): 577–599 (1998).

Raghu, P., et al., Unsupervised texture classification using vector quantization and deterministic relaxation neural network. In *IEEE Transactions on Image Processing*, volume 6, 1376–1387 (1997).

Rawlins, G. J., *Foundations of genetic algorithms*. Morgan Kaufmann Publishers (1991).

Rich, S. and V. Venkatasubramanian, Model-based reasoning in diagnostic expert systems for chemical process plants. *Computers and Chemical Engineering*, 11(2): 111–122 (1987).

Rivera, S. L. and E. J. Klein, Automatic classification of chromatographic peaks. In *Proceedings of American Control Conference*, volume 5, 3262–3266 (1997).

Roubos, J., et al., An evolutionary strategy for fed-batch bioreactor optimization; concepts and performance. *Journal of Biotechnology* (1999).

Rumelhart, D., et al., *Learnining Internal Representations by Error Propagation*. MIT Press (1986).

Sandler, S. I., editor, *Models For Thermodynamic And Phase Equilibria Calculations*. Marcel Dekker, University of Delaware, Newark, Delaware (1994).

Segovia, J. and P. Isas, Genetic programming for learning rule search in neural nets. *Neural Network World*, 8(2): 201–212 (1998).

Sharma, R., et al., Potential applications of aritificial neural networks to thermodynamics: Vapor-liquid equilibrium predictions. *Computers and Chemical Engineering*, 23: 385–390 (1999).

Shum, S., et al., An expert system approach to malfunction diagnosis in chemical plant. *Computers and Chemical Engineering*, 12(1): 27–36 (1988).

Simutis, R. and A. Lubbert, Exploratory analysis of bioprocesses using artificial neural network-based methods. *Biotechnology Progress*, 13: 479–487 (1997).

Smith, M., *Neural Networks for Statistical Modeling*. International Thomson Computer Press (1996).

Spears, W. M., et al., An overview of evolutionary computation. In *Proceedings of the 1993 European Conference on Machine Learning* (1993).

Stephanopoulos, G., Artificial intelligence in process engineering. current state and future trends. *Computers and Chemical Engineering*, 14(11): 1259–1270 (1990).

Storn, R., System design by constraint adaptation and differential evolution. *IEEE Transactions on evolutionary Computation*, 3: 22–34 (1999).

Strang, G., *Linear Algebra And Its Applications*. Harcourt Brace (1986).

Tabak, H. H. and R. Govind, Prediction of biodegradation kinetics using a nonlinear group contribution method. *Environmental Toxicology and Chemistry*, 12(2): 251–260 (1993).

Tsoukalas, L. H. and R. E. Uhrig, editors, *Fuzzy and neural approaches in engineering*. Wiley-Interscience (1997).

Umeda, T. and K. Niida, Process control system synthesis by an expert system. *Control Theory and Advanced Technology*, 2(3): 385–398 (1986).

Ungar, L., et al., Process modeling and control using neural networks. In *International Conference on Intelligence Systems in Process Engineering*, AIChE Symposium Series, 57–67 (1996).

Volk, W., *Applied Statistics For Engineers*. McGraw-Hill (1969).

Walas, S. M., *Phase Equilibria in Chemical Engineering*. Butterworth Publishers, University of Kansas (1985).

Wang, F.-S. and J.-P. Chiou, Optimal control and optimal time location problems of differential-algebraic systems by differential evolution. *Industrial and Engineering Chemistry Research*, 36: 5348–5357 (1997).

Wasserman, P., *Advanced Methods in Neural Computing*. Van Nostrand Reinhold (1997).

Weiss, M. A., *Data Structures and Algorithm Analysis in Java*. Peachpit Press (1998).

Werbos, P., *Beyond Regression: New Tools for Prediction and Analysis In the Behavioral Sciences*. Ph.D. thesis, Harvard University, Boston, MA (1974).

Whitley, D., et al., Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parellel Computing*, 14 (1990).

Willis, M., et al., System modeling using genetic programming. *Computers and Chemical Engineering*, 21 (1997).

Willis, M. J., et al., Artificial neural networks in process engineering. *IEE Proceedings, Part D: Control Theory and Applications*, 138(3): 256–266 (1991).

Wong, M. L., et al., Discovering knowledge from noisy databases using genetic programming. *Journal of the American Society for Information Science*, 51(9): 870–881 (2000).

Yang, M., et al., Neural network model for prediction of binary adsorption using single solute and limited binary solute adsorption data. *Separation Science and Technology*, 31(9): 1259–1265 (1996).

Yao, X. and Y. Liu, Evolving artificial neural networks. In *Proceedings of the IEEE*, volume 87, 1423–1447 (1997).

Yeun, Y. S. and K. H. Lee, Function approximations by coupling neural networks and genetic programming trees with oblique decision trees. *Artificial Intelligence in Engineering*, 13(3): 223–239 (1999).

Yu, H., et al., Combined genetic algorithm/simulated annealing algorithm for large scale system energy integration. *Computers and Chemical Engineering*, 24(8): 2023–2035 (2000).

Zhan, J. and M. Ishida, The multi-step predictive control of nonlinear siso processes with a neural model predictive control (NMPC) method. *Computers and Chemical Engineering*, 21 (1997).

Zhang, J., et al., Long-term prediction models based on mixed order locally recurrent neural networks. *Computers and Chemical Engineering*, 22: 1051–1063 (1998a).

Zhang, N.-W., et al., Statistical regression of binary vapor-liquid equilibrium data for ternary phase equilibrium predictions. *Fluid Phase Equilibria*, 147(1–2): 123–143 (1998b).

Zhang, X., Time series analysis and prediction by neural networks. *Optimization Methods and Software*, 4(2): 151–170 (1994).

Zhao, W., et al., Optimizing operating conditions based on ANN and modified GAs. *Computers and Chemical Engineering*, 24: 61–65 (2000).

Zhou, F., et al., Multiobjective optimization of the continuous casting process for poly (methyl methacrylate) using adapted genetic algorithm. *Journal of Applied Polymer Science*, 78(7): 1439–1458 (2000).

Zorzetto, L., et al., Process modelling development through artificial neural networks and hybrid models. *Computers and Chemical Engineering*, 24: 1355–1360 (2000).