A Neural Network Computer Model of the Hydrodynamical Flow in the River Medway Estuary at its Confluence with the River Thames

by

Lyn Hugh Rees

Submitted in accordance with the requirements

for the degree of Doctor of Philosophy

London Metropolitan University

Department of Computing, Communications Technology and Mathematics

June 2008

Declaration

The candidate confirms that the work submitted is his own and that the appropriate credit has been given where reference has been made to the work of others.

Acknowledgements

I would like to thank my supervisor Prof. Hassan Kazemian for his expert advice and guidance during this period of study.

I would also like to thank two visiting senior research fellows, London Metropolitan University - Dr. Brian D'Olier (my second supervisor) for his encouragement during this period and, Dr. Bryan Johns for his invaluable advice with regards to the hydrodynamics.

Not least but last, to my late wife Carol for her patience during this period prior to her untimely death in 2003 and to my present wife, Valerie, without whose encouragement this work would not have been completed.

Abstract

A neural network computer model of the hydrodynamical flow in the River Medway estuary at its confluence with the River Thames

A one dimensional, linearized, shallow water model finite difference scheme was developed to generate data representing both depth averaged velocities and depth fluctuations above or below the still water level in a river. After applying suitable boundary conditions based on the theory of characteristics, the model was then tested against another numerical model.

An artificial neural network (ANN) model for both depth and velocity with zero bottom friction was designed to use as a precursor to a full friction model. The model was extensively trained and tested over a 600 Km length, using generated data, to obtain information on the optimum structure of the neural network and various parameters. The model was then finally trained and validated over a 1200 Km length to avoid the danger of overfitting.

Using this frictionless model, it was extended to incorporate the effects of bottom friction. However, it was observed that the ANN was incapable of simulating rapid changes in the data close to the downstream boundary because of possible conflict between the non-linearized bottom friction and linearized boundary conditions. To overcome this difficulty, the standard bipolar activation function was replaced by a modified LeCun activation function. Subsequently, the neural networks were then re-trained and re-validated.

Prior to applying the ANNs to the confluence of the rivers Thames and Medway, the networks were tested for their adaptability to a variation of certain parameters. The models demonstrated good universal approximation capabilities when varying the imposed velocities, still water depths and friction coefficients.

Apart from minor discrepancies in generated depth and velocity data at the precise juncture of the two rivers, the networks showed more than adequate performance when simulating the flow in the two rivers.

Index

Chapter 1
Introduction

1.1 Historical Background

Since the estimation of river flows can assist in water management, protection from water shortages and flood discharges, such studies can have significant economic impact and the importance of such knowledge was realized by the inhabitants around rivers in early times. From earliest history, the recording by certain societies of river levels has been taking place and in fact according to Atiya et.al. [7], there are records of the flow level of the River Nile dating back to as far as 3000 B.C. Certainly in modern times, of all of the forecasting type of problems that have interested scientists, river flow is one of the earliest to have been considered.

As a consequence of many natural phenomena, surface water hydrographs (flow rate or flow discharge against time) of rivers exhibit large variations so that a common approach for interpreting and extending a stream flow record is to fit the observed data with some form of deterministic or statistical model. However, such models may not necessarily represent the flow process adequately, since such models are based on many simplifying assumptions about the physical processes that influence river flows.

Since dendritic (tree like) river systems are rather complex, the modelling of these systems has at present, involved both rainfall-runoff and river flow routing simulations. Of the conventional river routing models that have been developed, most are SISO models (single input - single output, i.e. one input from the upstream and one output from the downstream). Examples are the hydraulic river routing model derived from the St. Venant equations, Mujumdar [81] and hydrological models such as the Muskingum model, Chadwick and Morfett [19]. Unlike their hydraulic counterparts, hydrological routing models not being distributed models, frequently lack the flexibility to predict flows at multiple ungauged sites along a river. Hence, to develop a model for dendritic river systems that are essentially

MISO in nature (multiple input – single output), rainfall-runoff models need to be integrated with hydraulic routing models.

Unfortunately, to develop such integrated models, a large amount of data from numerous sites along the river reaches is needed to provide sufficient representation of the hydraulic characteristics that may change over the length of the river. This is both time consuming and expensive, so that some recent research has focussed on establishing linear MISO models. MISO models can generally be expressed in terms of two explicit characters: One parameter, referred to as a cascade character, represents different hydrographs at separate cascade points along the river, and a second parameter that describes the multiple inputs. As previously mentioned, a hydrograph is a time record of discharges of rivers or watershed outlets, the main input to a watershed being typically rainfall whilst output is the streamflow. Thus a hydrograph is a time representation of how a watershed responds to rainfall.

However, these models are all linear models and hence when operated in real time, the responses (the watershed output response to rainfall) tend to become unstable. That is, the *predicted* model discharges (the flow rates) are inconsistent with the actual measured values. Consequently, current research is focussing on developing a non-linear modelling methodology to represent the dendritic river systems that are by their very nature, highly non-linear and non-stationary. In fact, much research on these problems has been conducted at the University of Bristol by various researchers such as Hammond and Han [39], Han et. al. [41] and Yang and Han [110].

The practice of rainfall-runoff modelling, Chadwick and Morfett [19], was in essence the result of the availability of extended records of rainfall and other climatic data, from which stream flow data could be obtained. Whilst physically-based (or conceptual) models may be important to understand the underlying hydrological processes, in many situations where the primary concern is to be able to make accurate predictions at specific locations without necessarily understanding these processes, machine learning models can be utilized. In other words, a 'black box' approach can be used.

1.2. The Application of Artificial Intelligence

The method of river flow forecasting has traditionally been addressed using techniques such as nonlinear regression, Kachroo et. al. [56] and Kachroo et. al. [55] or computer models involving approximation techniques such as finite differences, Lin and Falconer [66]. However, artificial neural networks (ANNs) have the advantage of being able to use field data directly without any simplification, unlike regression analysis wherein some assumption has to be made *a priori* as to the form of the equation. Further, neural networks are capable of executing parallel computations and can simulate non-linear systems (a common feature of hydrodynamic problems) that are difficult to describe using traditional modelling methods.

Although ANNs have been in existence for approximately sixty years, McCulloch and Pitts [77], it is only in the last two decades that there has been a resurgence of interest in them. ANNs are in essence an attempt to replicate in a computerised manner the way the human brain solves complex problems. They have been applied successfully to pattern recognition, optical character recognition, voice recognition, petroleum surveys, short term rainfall forecasting, waste water treatment of toxic chemicals, signal processing, oceanographic, Agrawal and Deo [3], hydrodynamic, Dibike and Abbott [27], hydrological, Cigizoglu [23] hydraulic engineering, Sonnenborg [98], and statistical problems such as non-linear regression and time series analysis.

Most ANNs used in forecasting are of the multilayer network type trained using the backpropagation algorithm, and this is certainly true for river flow forecasting. In addition, in this particular field of interest, many of the models only give a one-day ahead forecast which is simpler to calculate but less useful than a ten-day ahead or one-month ahead forecast.

Atiya et. al. [7], observed (at least at the time of publication), that there seemed to be only one application of neural networks related to the problem of river flow forecasting, namely, that for the Huron River in Michigan, Karunanithi [57]. A literature research by this researcher has found only four papers explicitly dealing with river flow modelling using neural networks, *prior* to the publication of their paper, namely Dibike and Abbott [27],

Karunanithi et. al. [57], Thirumalaiah and Deo [101] and Zealand et. al. [111]. However, in the last decade, the application of neural networks to this field has proliferated, there being, discounting any related to support vector machines (SVMs), at least nineteen published papers since that of Atiya. Additionally, it would appear that the application of support vector machines (a new tool from the Artificial Intelligence Field) to this type of problem is at the same stage that neural networks were a decade ago, since the same literature research has so far revealed only two applications of support vectors to river flow forecasting. ANNs are discussed in Chapter 2 whilst SVMs (since they are not employed within this research) are instead dealt with briefly in appendix D for possible use by other researchers.

From the literature surveyed so far, it would appear that traditional computer techniques are prone to difficulties when trying to model multiple input flows with a single output flow. This is almost exactly (ignoring tidal reversal flows) the type of regime that is the subject of this research, since the Thames-Medway confluence is a multiple input site. The situation is complicated even further if tidal reversal is included since the MISO scenario oscillates between MISO and SIMO (single input - multiple output). Hence, it is proposed to use the techniques of ANNs to develop a computer model of this area. Unfortunately, as there is little available observed data in this region, it is proposed to first develop a suitable one-dimensional finite difference scheme based on the shallow water theory. From this, artificial data will be generated which will then be used to train a suitable neural network. Most importantly, to ensure the data and models are as realistic as possible, the effects of bottom friction will be included. To this researcher's knowledge, the inclusion of such effects in a neural network model as described in particular in Chapter 5, has not been performed before. A detailed summary of work conducted by researchers in this particular field of hydrodynamics is included at the beginning of Chapter 3.

1.2.1 Justification for the use of artificial neural networks

As has been described, river modelling has to overcome difficulties from sources such as:

- Lack of complete measured data or incomplete data
- Very complicated hydrodynamical equations to be modelled analytically
- Multiple input – single output and single input – multiple output flows

- Inability to model these hydrodynamical equations particularly where turbulence is involved without some form of averaging of the measured quantities with hence an attendant loss of accuracy.

Since neural networks can simulate directly from the given data, without any *a priori* equations, they can overcome most if not all of these difficulties.

1.3. Aim and objectives of the research

The aim is to develop an artificial neural network (ANN) incorporating bottom friction to model flow parameters such as velocity and depth in the area of confluence of the rivers Thames and Medway. To achieve this, the research has the following objectives to pursue:

1. Develop a finite difference model based on the one dimensional shallow water equations to generate data to use for training and testing of the ANN.
2. Develop and test different ANNs with zero bottom friction to obtain information on the optimum network structure and parameters.
3. Using the zero bottom friction model as a template, extend the network to include the effects of bottom friction.
4. Test the universal approximation capabilities of this bottom friction model with a variation in the values of the upstream and downstream imposed velocities, still water depth and coefficient of bottom friction.
5. Finally, apply this model to the confluence of the rivers Thames and Medway.

1.4. Outline of the thesis

In **Chapter 2** the fundamentals of neural networks are discussed in preparation for their use in Chapters 4 and 5. A description of a simple artificial neuron and its analogy with the biological network is provided. Activation functions and various types of neural network architecture are described, followed by a discussion of the various methods of training the network. The advantages and limitations as well as the history of neural networks are perused.

Chapter 3 essentially focuses on the development of two finite difference schemes, one without bottom friction and the other with it. An introduction to shallow water theory is given from which the finite difference equations can be developed. Stability, convergence

and compatibility of these equations are analysed and the necessary boundary conditions are discussed. The generation of artificial data for both models, with and without bottom friction, is developed and discussed extensively.

Chapter 4 consists of a discussion of the development of a neural network to model the river flow where there is zero bottom friction. A description of the chosen network architecture and the reasons for this choice are described extensively. The backpropogation algorithm is the selected method of training and reasons for this selection are discussed therein. The algorithm is described in detail. Two networks are actually developed since both the depth and velocity of flow need to be modelled. After discussing several measures of error, training of the depth network is described. This involved evaluating the optimal parameters such as the number of hidden neurons as well as the learning rate and momentum terms. A similar procedure was used to evaluate these parameters for the velocity network. Both networks were trained over an artificial 1200 Km length of river of fixed depth in ebb tide mode. The networks were then validated using (unseen) flood tide data. Finally, the chapter concludes with a comparison of the two networks with the numerical models of another researcher, Johns [51].

Chapter 5 is concerned with probably the most important objective of this research. It describes, to this researcher's knowledge a completely novel use of neural networks. That is, the development of a neural network model for both depth and velocity, but unlike the previous chapter, to include the effects of bottom friction. For consistency, the procedure followed that of the zero friction model with training performed over 1200 Km in an ebb tide regime. This was followed by validation using generated flood tide data. It was observed that the model displayed inadequate performance near to the estuarine boundary as a consequence of the nonlinear bottom friction effects. A re-appraisal of the architecture and parameters was conducted resulting in the application of an alternative activation function, in this case, a novel use of the LeCun activation, LeCun [64], LeCun [63] and LeCun et. al. [65]. The networks were then re-trained and re-validated and compared against the results obtained using the original standard bipolar activation function. Having finalized the models, comparisons were then made between variations of the different flow parameters to

evaluate the performance of the two finalized networks. Finally, the research focussed on the application of these two models to the area encompassing the confluence of the rivers Thames and Medway.

Chapter 6 is a discussion of the results and conclusions with recommendations as to possible avenues of further research. It is noted that objectives 1 to 4 were achieved satisfactorily. Objective 5 was partially successful in that there were some minor discrepancies in the generated data at the juncture of the two rivers. This apart, the actual neural network simulations themselves were successful.

Several appendices follow:

Appendix A is a listing (nomenclature) of the various terms used throughout the thesis.

Appendix B is a mathematical description of the finite difference schemes involved.

Appendix C is a complete listing of the code of the various neural network programs.

Appendix D contains a description of support vector machines that may prove useful for other researchers following this avenue of investigation.

Appendix E is a description of the Thames/Medway confluence as well as a brief history of its formation.

Appendix F is a detailed plan of the area showing the bathymetry of the area of confluence.

Chapter 2
Application of Artificial Intelligence

2.1 Introduction to Artificial Neural Networks

This chapter is essentially a description of the fundamentals of artificial neural networks (ANNs). The development of neural networks has its roots in the study of the human brain and as a consequence, many of the technical terms have biological equivalents. It was intended that these computational structures would emulate the brain so that tasks at which brain was very efficient at processing, the neural network should be able to exhibit similar capability. Conversely, any processing tasks that the brain could not do very well, a neural network should likewise perform poorly on them. To differentiate between the biological and the artificial (computing) networks, the latter are often referred to as artificial neural networks, neurocomputers or connectionist networks. To put this into perspective, a brief discussion follows on the human brain and its artificial counterpart.

2.2 The Brain - the biological neural network

The brain consists of approximately 10^{10} neurons (nerve cells) and an estimated 6×10^{13} neural connections since each neuron may have on average, connections with up to about 10^4 different neurons. It is a highly complex, non-linear parallel computing system with the ability to reorganize its structural components (the neurons) to perform certain computations many times faster than existing digital computers. Such examples are pattern recognition (human vision) and analogue signal analysis (bat sonar).

At birth, the brain is structured with the ability to build its own 'rules', that is to learn through experience, with the most significant development taking place in the first two years. A property called 'Plasticity', Haykin [43], permits the development of the nervous system to adapt to its environment. In the brain, the neurons are the information processing units. Fig.2.1 that follows, is a simplistic representation of a biological neuron (nerve cell).

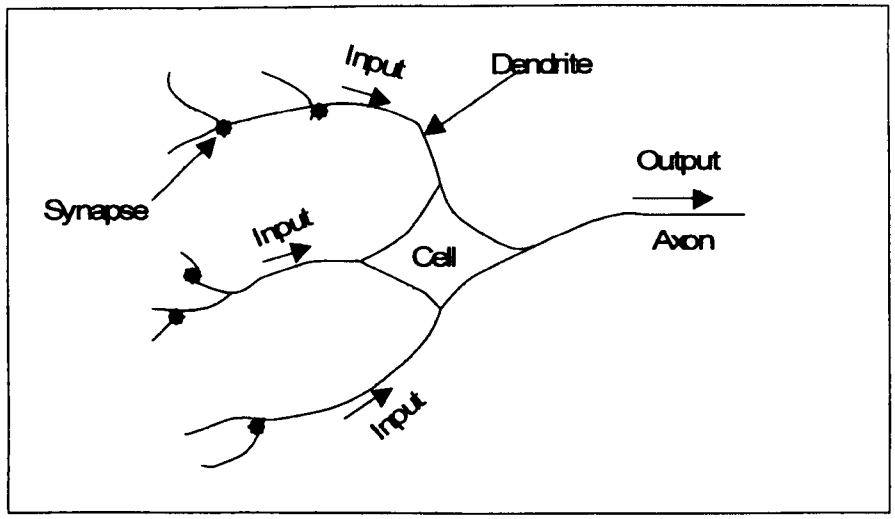

Figure 2.1:Biological Neuron

Inbound neuron connections are called dendrites whilst the outbound ones are called axons. Where the axon of one neuron connects with the dendrite of another neuron there is a small biochemical 'connection' called a synapse. When electrical signals travelling down the axons arrive at the synapse they are converted to chemicals. Much like a capacitor in electrical systems, the biochemical level represents the desire to fire, i.e. 'excitation', (the opposite being referred to as 'inhibition') on the neurons that follow.

When this level reaches a certain point (from the sum of all the other incoming connections), the neuron converts these chemicals at the synapse into one big electrical pulse (sometimes referred to as action potentials or spikes) and fires it down its axons. This pulse is then converted to chemicals at the other neurons' synapses and in this way, the cycle continues. Interestingly, as opposed to firing continuous signals of varying strength over short or long periods of time, the neuron sends a pulse and resets the synapse. Since there is a mesh of interconnected neurons, this results in a storm of firing neurons that continues all the time. This is referred to as 'brain activity'.

If each neuron is regarded as a switch or microprocessor, the brain is quite slow in comparison to a modern computer. Neurons fire about once every 60th or 70th of a second, whereas silicon based devices work at speeds in orders of nanoseconds. However, it is the massive parallelism of the brain that enables it to compete with the silicon microprocessor, at least for the moment. Further, the brain has some key self-modifying abilities that are

difficult to replicate in a computer. New connections are continuously being made and broken between the neurons, a process that is often referred to as self modification (or 'learning'). This ability also gives rise to the property of 'fault tolerance' wherein nearby neurons can take over the duties of a damaged neuron. Although there are many definitions of an artificial neural network, one of the most well known is that attributed to Haykin [43]: *"A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects: Knowledge is acquired by the network from its environment through a learning process. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge."*

Biological – the brain	Artificial Neural Network
Soma (nerve cell or neuron)	Node
Dendrite	Input
Axon	Output
Synapse	Weight
Slow speed	Fast Speed
Many Neurons (thousands of millions)	Few neurons (up to hundreds of thousands)

Table 2.1: relationship between the brain and an artificial neural network.

2.3 Simple Artificial Neuron

An artificial neuron can be linear or non-linear the latter property being needed if the physical phenomena to be modelled are non-linear (e.g. speech).

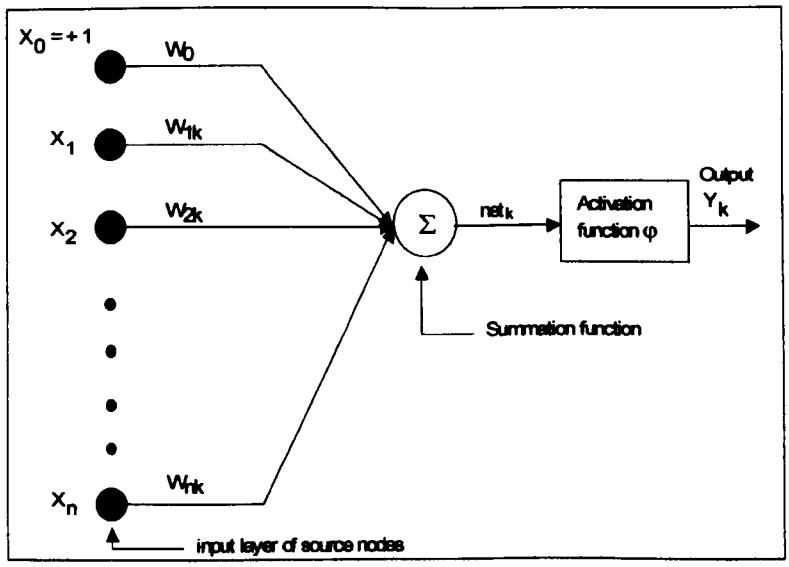

Figure 2.2: Artificial Neuron

Fig. 2.2 is a representation of a simple non-linear model of an artificial neuron and depicts four basic elements:

2.3.1 The input layer of source nodes

This is the raw data and if it is not digital, but analogue input such as text, sound or graphics, it will need to be pre-processed first into digital (numeric) equivalent form generally in the format of binary numbers. In figure 2.2, x_n is the input signal associated with the n^{th} synapse for a particular neuron, in this case the k^{th} neuron.

2.3.2 A set of synapses (connecting lines)

Each has a corresponding weight (strength) of its own. Here the weight w_{nk} refers to the weight for the x_n input to the k^{th} neuron. These weights may be modified during training according to the ANN's specific topology or learning rules. It should be noted from figure 2.2, that there is a weight w_o known as the 'bias' that is applied to the input x_o, the latter being always = +1. This bias unit x_o is optional but it has the effect of lowering or increasing the net input of the activation function dependant upon whether the net input is positive or negative respectively and so sometimes helps the network to converge to an acceptable level of accuracy.

2.3.3 An adder (or summation function)

Sometimes referred to as a linear combiner, this function sums the weighted inputs. For example, with a value of always +1 for the input x_o and some bias w_o, then the net input to the k^{th} neuron

$$\text{net}_k = w_0 + \sum_{i=1}^{n} w_{ik} x_i \qquad \textbf{(2.1)}$$

can be written more succinctly as

$$\text{net}_k = \sum_{i=0}^{n} w_{ik} x_i \qquad \textbf{(2.2)}$$

It should be noted that this simple summation function may well be more complex dependant on the nature of the network architecture and paradigm. In addition to basic product sums, this function may well involve the use of minimum, maximum, majority, average, normalized or Boolean logic values.

2.3.4 An activation function

Frequently referred to as a squashing function or transfer function, Haykin [43], here the result of the summation function determines the activation level of the neuron and dependant upon the latter, whether or not the neuron produces an output. It should be noted that this dependency may be linear or non-linear. The purpose of the activation function is to limit the amplitude of the output to within a certain reasonable range, usually [0,1] or [-1,1], a necessity if the output is passed on to another neuron in the network. There are many types of activation (transfer) functions in use, the most frequently used ones being:

Threshold Function

$$y_k = \phi(net_k) = \begin{cases} 1 & if\ net_k \geq 0 \\ 0 & if\ net_k < 0 \end{cases} \qquad \textbf{(2.3)}$$

which is essentially a Heaveside type function, Abramowitz and Stegun [2], where y_k is the output of neuron k, ϕ is the activation function and net_k is the weighted sum of the inputs.

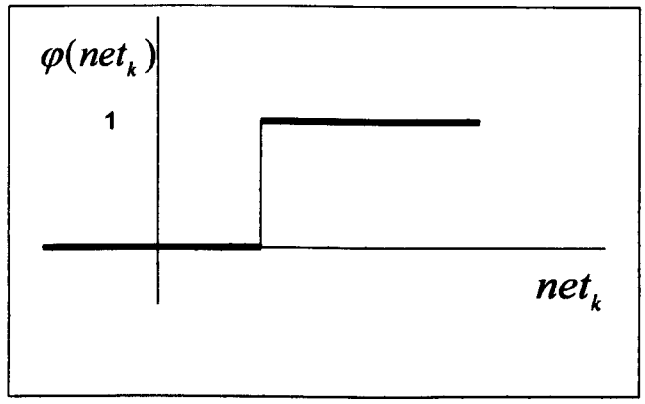

Figure 2.3:Threshold Function

Sigmoid Functions

An 'S' shaped curve with minimum and maximum values at the asymptotes, it can be represented by the logistic function:

$$y_k = \varphi(net_k) = \frac{1}{1+\exp(-D \times net_k)}, \qquad y_k \in [0,1] \qquad \textbf{(2.4)}$$

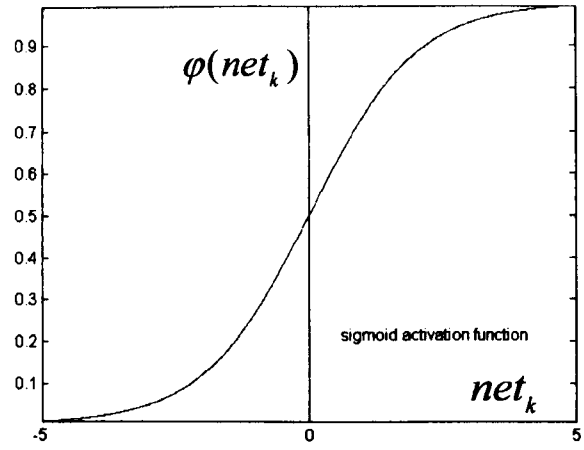

Figure 2.4: Sigmoid Function

where usually D = 1.

Tanh Function (or hyperbolic function or bipolar sigmoid function)

$$y_k = \varphi(net_k) = \frac{2}{1+\exp(-2 \times net_k)} - 1, \qquad y_k \in [-1,1] \tag{2.5}$$

which has a similar S shaped graph to the sigmoid. The outputs are centred around zero which some researchers have noted results in faster training due to the better numerical conditioning, Sarle [92].

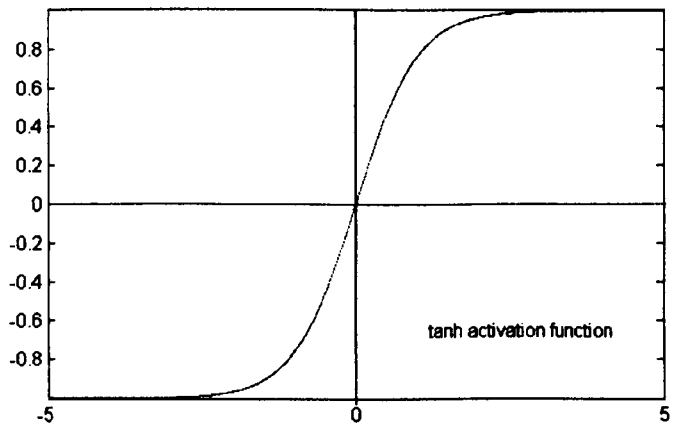

Figure 2.5: Tanh Function

Gaussian Function

$$y_k = \varphi(net_k) = \exp(-(net_k)^2), \quad y_k \in (0,1], \quad net_k \in (-\infty,\infty) \tag{2.6}$$

Other sigmoid functions

$$y_k = \frac{net_k}{1+|net_k|}, \qquad y_k \in [-1,1]$$
$$y_k = \frac{net_k}{2(1+|net_k|)} + 0.5, \qquad y_k \in [0,1] \tag{2.7}$$

Sarle [92], both of which are quick to compute but approach the extreme (target) values slowly.

Linear Activation Function

$$y = Dx \tag{2.8}$$

This is sometimes referred to as the identity activation function.

When the inputs are signals that could be represented by a Fourier series, Picton [85] notes that some researchers have claimed improved performance if the transfer function is replaced by a trigonometric one such as sin(x) or tan(x). In regression problems where the output is attempting to match a real number rather than a yes/no decision as in classification problems, the transfer function is often a linear one since this is purported to model the system better.

Picton [85] further notes that if the inputs could be represented by a Fourier series, a possible candidate for an activation function could be sin(x). The primary purpose of activation functions, in contrast to linear transfer functions, for the hidden units (discussed later) is to introduce a non-linearity into the neural network. A neural network without any non-linearity is representative of simple, weak perceptrons that contain just inputs and outputs. He [Picton] also further comments that any non-linear function (except those of polynomial form) will suffice for backpropagation (discussed fully in Chapter 4) provided it is differentiable and preferably bounded. Picton also adds that some authors have suggested that arctan might be better than tanh. Thus, if a non-linear property such as bottom stress (the subject of Chapter 5) is to be modelled, then a network using the identity activation function will not be suitable.

2.3.5 Scaling

It should be noted that if the raw data (be it input, target or both) needs to be scaled to fit the range of the activation functions, then the following formulae can be used:

$$x_s = \frac{x_i - 0.5(x_{\max} + x_{\min})}{0.5(x_{\max} - x_{\min})} \text{ such that } x_s \in [-1,1] \text{ for the bipolar sigmoid} \quad \textbf{(2.9)}$$

or

$$x_s = \frac{x_i - x_{\min}}{(x_{\max} - x_{\min})} \text{ where } x_s \in [0,1] \text{ for the sigmoid} \quad \textbf{(2.10)}$$

and x_s, x_i are the scaled and unscaled values of the raw data respectively, whilst $x_{\max}, x_{\min}$ are the maximum and minimum values of these sets of raw data. The formulae can be 'inverted' to rescale the data back to the original values after training. However, in many cases such as a multilayer perceptron (MLP) where the input data are combined linearly, theoretically at least, Sarle [92], it is not strictly necessary to do so.

2.4 The Artificial Neural Network

Artificial Neural Networks (ANN) are software programs (or sometimes dedicated firmware burnt into a silicon chip) that attempt to allow the computer to emulate the process by which the human brain stores and recalls information. Not surprisingly then, the software system consists of many thousands of (or even millions of) neurons (data cells or nodes) that, when an input stimulus is received, send output signals to the nodes to which they are connected. Each connection has a value, or weight, which determines the 'strength' of the signal sent to a receiving node. When several nodes are stimulated simultaneously, the learning algorithms are able to modify the weight values accordingly. Neural networks can be classified in terms of type based on the following:

Learning Methods

- Supervised
- Unsupervised

Connection Type

- Feedforward (sometimes referred to as static)

- Feedback (sometimes referred to as dynamic)

Topology (architecture)

- Single layer
- Multilayer
- Recurrent
- Self-Organized (SOM)

Applications

- Classification
- Function approximation
- Prediction

2.4.1 Neural Network Architectures - Feedforward

Each ANN is a collection (clustering) of artificial neurons (the processing elements) grouped in layers as depicted in fig. 2.6. This is an example of a single layer network so called because the single layer refers to the output layer of neurons depicted by the hollow circles and not the source nodes (solid circles). It is within the hidden (where they exist) and output layers that the computation is performed. Throughout the rest of this chapter, the convention is that the nodes/layers where computation is performed will be represented by hollow circles in the figures that follow.

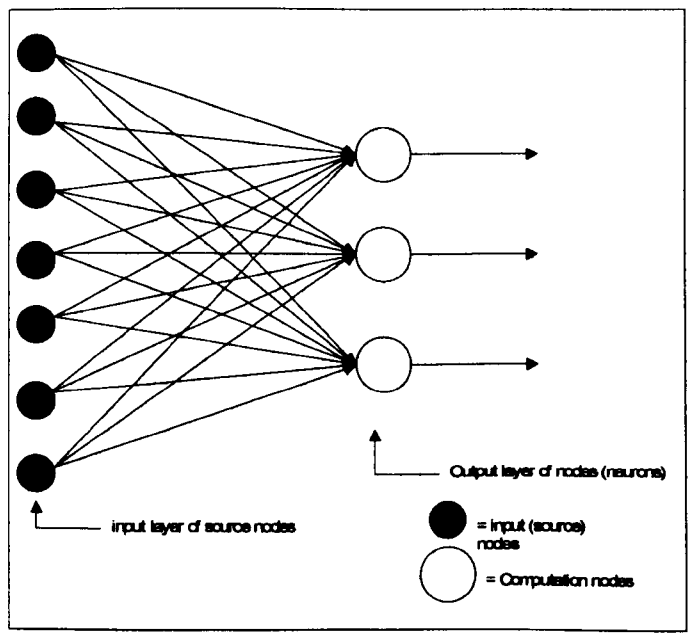

Figure 2.6: Single layer feedforward network

Unlike the single layer network above, it is more usual to have a hidden layer in the architecture such as in fig. 2.7 that follows, Gurney [38]:

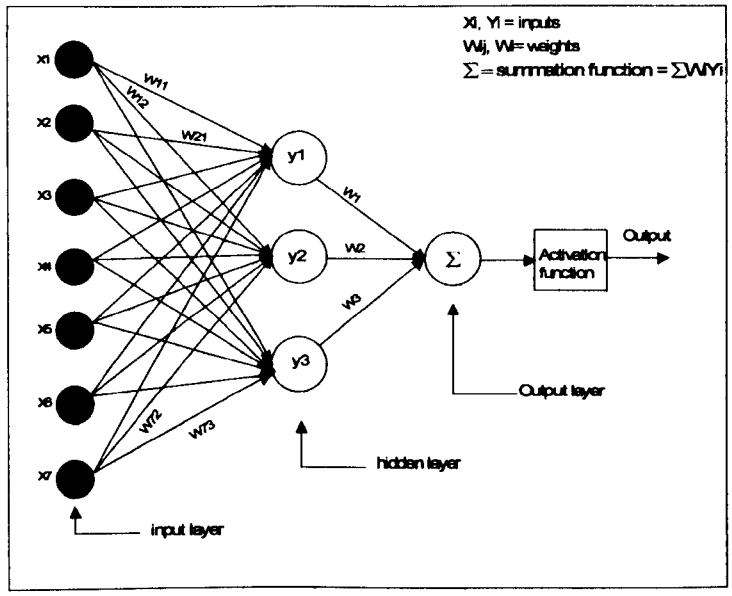

Figure 2.7: Multilayer feedforward network

The example shown, a multilayered network, depicts an ANN with one hidden layer and so would sometimes be referred to here as a 7-3-1 network. Unfortunately, this can be ambiguous as some authors refer to the weights as layers. To avoid ambiguity, it is better to refer to this example as an ANN with one hidden layer. There are three main layers in this example: the input layer, the intermediate (or hidden) layer and the output layer. This is a typical example of the multilayered perceptron (MLP). The addition of a hidden layer(s) enables the network to extract higher-order statistics and is particularly important when the size of the input layer is very large, Sarle [92].

Different architectures have different numbers of neurons in both the input and output layers from the example shown above and further, may have more than one hidden layer (fig. 2.8), Gurney [38]. Usually though, one hidden layer is sufficient (see Section 2.4.3).

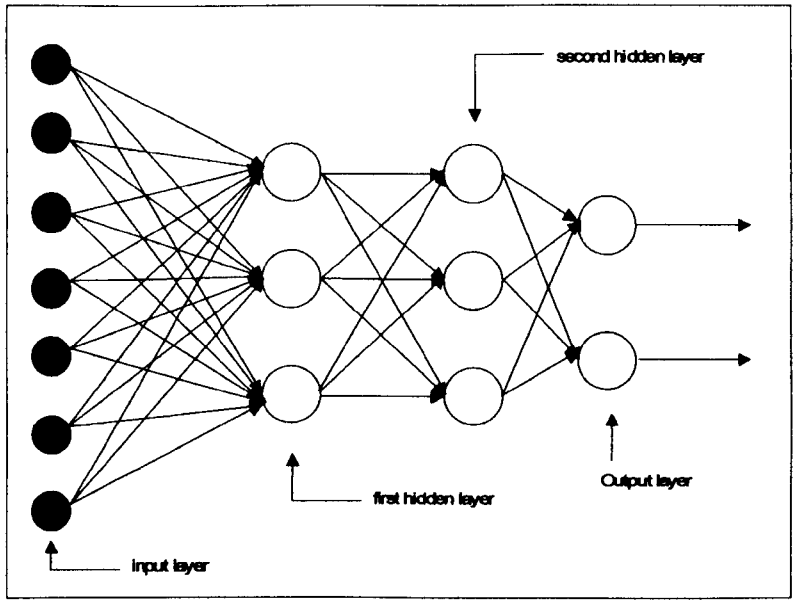

Figure 2.8: Feedforward network with two hidden layers

To complicate matters even further, some architectures do not have full connectivity as shown in fig. 2.9, Gurney [38]. In this example, three of the input neurons are not connected to all of the nodes in the hidden layer. This might occur where the designer has some *a priori* knowledge of the problem domain.

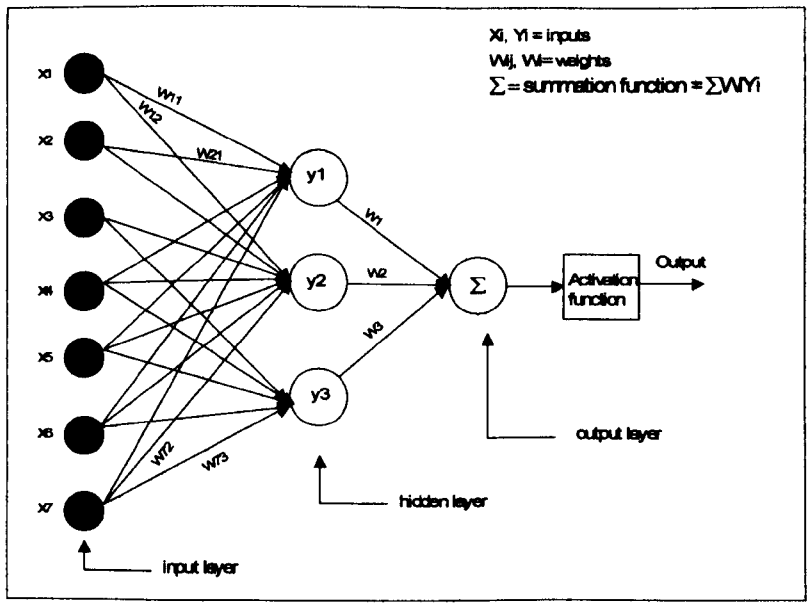

Figure 2.9: Network not fully connected

A final complication is that the network might be non-layered (where one or more input neurons bypass the hidden nodes and connect directly to the output layer), Gurney [38].

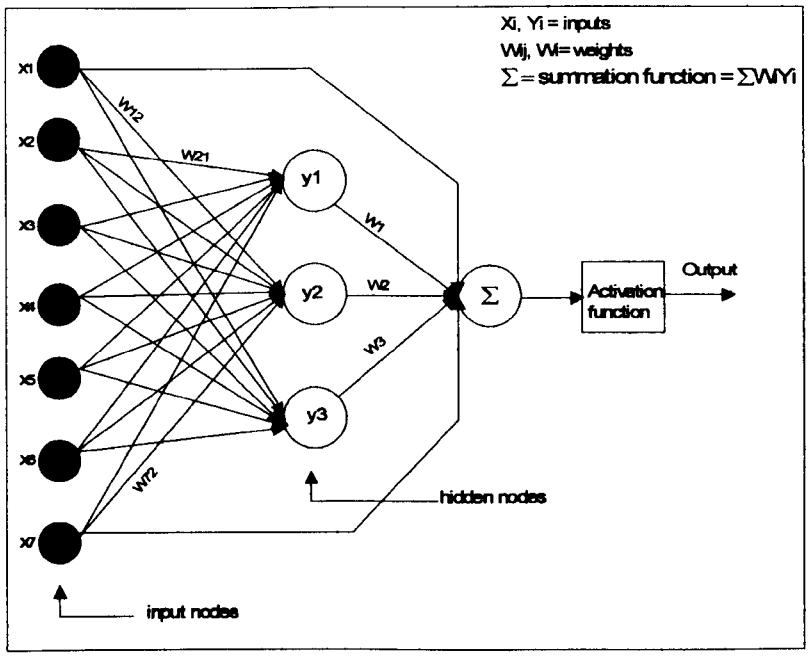

Figure 2.10: Non-layered network

This is exemplified in fig. 2.10 above where two of the input neurons not only connect to some of the hidden nodes but also have direct connections to the output node. All of the architectures so described, figs. 2.6 to 2.10, are examples of 'feedforward' structures in that they do not have any loop back mechanism to the input layer, in other words, neuron outputs are always fed forward to subsequent layers. There are two main classes of single layer feedforward networks, namely Rosenblatt's [90] perceptron and Widrow and Hoff's [109] adaptive filters. The former is based on the non-linear neuron model of McCulloch and Pitts [77] and is ideal for pattern classification whilst Widrow and Hoff's model (using the LMS algorithm) has been applied in fields such as communications, radar and seismology.

Feedforward networks are useful for solving pattern recognition, classification and generalization problems such as for example, quality control and loan evaluation. Examples of this type of network are: single layer perceptrons, multilayer perceptrons, radial basis functions, support vector machines, committee machines and stochastic machines. Radial basis function networks are non-linear layered networks that are particularly useful for solving multivariate statistical interpolation and complex pattern recognition problems. Examples of stochastic machines are Boltzmann and Helmholtz machines whilst Committee

machines are based on the 'divide and conquer' principle with complex tasks being split up into smaller ones before applying a neural network solution to it.

Typical algorithms used for training purposes of feedforward networks are backpropagation, Madeline III and Least Mean Squares, Sarle [92]. The backpropagation algorithm is discussed more fully in Chapter 4.

2.4.2 Neural Network Architectures - Feedback

In contrast to the previous section (cf. Section 2.4.1), fig. 2.11 that follows is an example of a feedback (or recurrent) architecture, Callan [17] and Skapura [96], where at least one or more output nodes has a direct feedback to one or more input nodes. It is implemented using a time step delay procedure as represented by the Z^{-1} notation in the figure.

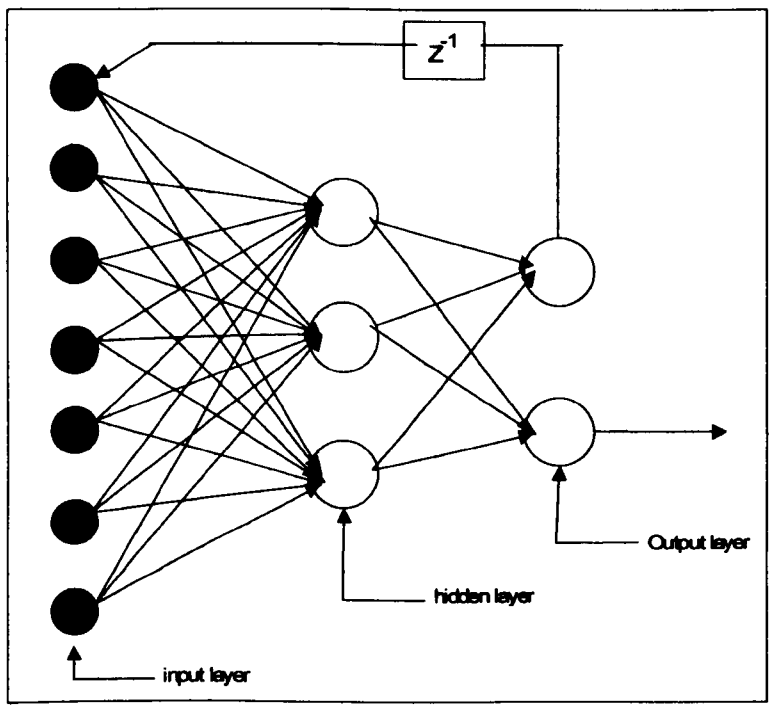

Figure 2.11: Recurrent network

Recurrent networks are useful for solving dynamic time dependent problems such as sales forecasting, and process analysis, examples of this type of network being Hopfield Nets and Recurrent Multilayer Perceptrons. Typical algorithms used for training purposes of recurrent networks are: backpropagation through time, true time and real time recurrent learning.

2.4.3 Number of hidden layers

Kolmogrov's Existence Theorem, Lippmann [67], provides a useful guide to the required number of hidden layers, that effectively states that any three-layer network with n(2n+1) nodes where n is the number of nodes in the input layer, should suffice. This was further supported by the work of Lapedes and Farber [62], who argued that no more than two hidden layers were necessary, although they did qualify this argument by indicating that this would not necessarily be the most efficient arrangement. However, two hidden layers can lead to more problems with local minima. Further research, Hornik [45], supports the argument that only three layers are needed. It provided a theoretical proof that three layer perceptrons with a sigmoid output function are universal approximators, the accuracy of the approximation being dependant on the number of neurons in the hidden layer.

2.4.4 Number of neurons in the hidden layer

Although there are no specific formulae for the determination of the number of required hidden units, the number is often obtained by experimentation. Some researchers have used the guideline that if n is the number of input neurons and m the number in the output layer, then the required number is $\sqrt{nm}$. Other researchers have used the guide $\ln M$ where M is the number of different input patterns. Needless to say, too few hidden units and the network will fail to train correctly due to *underfitting*, that is, the network is insufficiently complex for the problem at hand, whilst too many hidden units and the network will fail to generalize (it tends to memorize). In other words, it is suffering from *overfitting*. Too many units also of course lead to increased computational effort. However, it is possible to remove superfluous units by examining the weight changes during the course of training. Two or more inputs usually require more than one hidden units to realize a fit to a wider spectrum of functions.

2.5 Training the neural network

As to the required number of patterns for training, there is no rigorous rule. However, various researchers, Sarle [92], use the following guidelines:

- Number of training patterns should be greater than ten times the number of inputs
- Number of training patterns should be greater than thirty times the number of weights

Once a network structure has been designed for a particular application, the learning process or training can be implemented. This essentially consists of three main parts:

- compute the outputs
- compare the outputs with the desired target values
- adjust the weights and repeat the process

Learning algorithms are used to modify these weights in an orderly fashion to achieve a desired objective of 'accuracy'. At present there are more than a hundred learning algorithms available for different network configurations and applications, the learning process itself being classified as supervised or unsupervised. One of the most widely used algorithms for training a multilayer perceptron is the 'Backpropagation algorithm' (or Generalized Delta Rule), which as previously mentioned, will be employed in Chapter 4. The backpropagation algorithm, Haykin [43], Picton [85], Tarassenko [99] et. al., being essentially a gradient descent process wherein the weights are adjusted in a particular way, uses two parameters called the learning rate and momentum term, the latter being optional. After learning is complete, testing takes place and is a typical 'black box' scenario usually with results better than 70% accuracy. If the accuracy is insufficient, the network can be retrained or quickly reorganized.

2.5.1 Selection of weights

There are no fixed rules for choosing the weights. However, if a training algorithm such as backpropagation is used, the initial weights should ideally be chosen randomly. This is because the magnitude of the errors backpropagated are proportional to these weights and as

such the weights would be updated by the same amount. Further, to avoid saturation of the activation function, the initial weights should be distributed within a small range of values.

2.5.2 Learning Rate

The learning rate parameter η controls the amount by which the weights are updated during each backwards pass of the backpropagation cycle. The value of this parameter determines the speed of convergence to a solution but if not chosen wisely, there is a possible tendency for the training to become entrapped in local minima. There being no precise formula for the optimum value of this parameter, it depends on the particular problem at hand and is usually obtained by experimentation. Where these minima are broad, indicated by small gradient values, then larger values of this parameter lead to faster convergence. However, when these minima are narrow, indicated by large gradient values, smaller values need to be chosen to avoid overshooting the solution and becoming trapped in local minima. Unfortunately, this latter argument results in more computational effort. Sometimes, researchers begin with a chosen value for the parameter and increase it or decrease it (as in *stochastic approximation* or *adaptive learning*), Sarle [92], carefully as the training progresses. Frequently, values between [0.001, 10] are chosen but more usually between [0.01, 1]. Some researchers use $\eta < \frac{1}{n+1}$ where n = no. of input neurons and when the bi-polar sigmoid is used, the value of the training parameter can be doubled, Picton [85].

2.5.3 Momentum Term

The function of the (optional) momentum parameter α, which usually $\in [0.1, 0.8]$, is to aid the convergence of the backpropagation algorithm. It uses a fraction of the most recent weight adjustment and tends to keep the weight changes going in the same direction as well as smoothing out the gradient descent path. This prevents wild oscillations about the error surface.

2.5.4 Local Minima

One of the difficulties inherent in training a network using backpropagation is, as previously noted, the propensity to get 'trapped' in local minima. A two dimensional error surface with one local minimum, is shown below in the following figure:

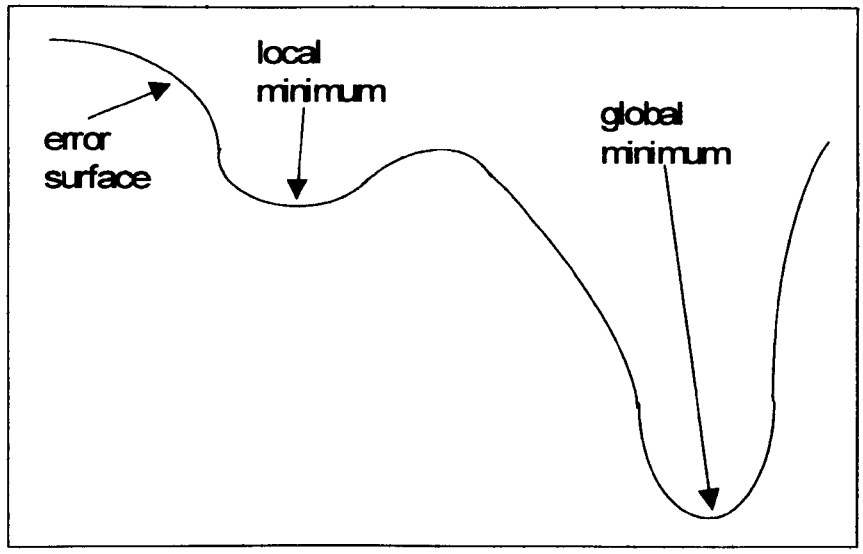

Figure 2.12: Error Surface

As the backpropagation network (BPN) is a pure gradient descent process, the error signal can easily become trapped in the local minimum, since only steps with a negative gradient can be taken on the error surface. This can sometimes be remedied by using a different learning parameter, a different momentum term (if present) and/or repeating the training with different random weights. The latter option is equivalent to starting the network at a different point on the error surface.

Dependant upon the problem being considered and the available data, training of the network can take place using two different strategies:

2.5.5 Incremental Updating

In this method, weight adjustments are made after the presentation of each single pattern with the patterns themselves being chosen randomly from the training set.

2.5.6 Epoch based learning

In this alternative strategy, sometimes referred to as 'cumulative weight adjustment', the complete set of training patterns are presented to the network, an average error calculated from this and then weight adjustments made accordingly. Although computationally more efficient, it is more susceptible to the problems of local minima. It is also more demanding of computer resources.

2.5.7 Training Set

Whilst training, the error (referred to as the error function) between the actual and desired (target) outputs is measured with the aim of reducing this error to an acceptably low level by adjusting the weights. Once training has 'finished', the network is presented with the test set comprising data it has never seen before and its performance measured. It has been observed that a neural network never performs on a test set as well as it did on the training set, a phenomenon know as 'over-training' wherein the network is said to have overfitted the training data. It is classifying the training set very well but classifying the test set rather poorly. In other words, it is not generalizing very well.

2.5.8 Early Stopping

This problem of 'overfitting' can be overcome by splitting the original data set into three parts rather than two as mentioned above (cf. Section 2.5.7). The training set is used to measure the error between the output and the target data and reduce it by adjusting the weights. However, whilst training, the test set is also presented to the network and the error noted, but no adjustment of the weights is made. During this process of training, as for the training set, the error in the test set should also fall but be higher than that of the training set. When the error in the test set data stops falling (perhaps even starts to rise), the point at which overtraining has started has been reached. If training is stopped at this point, overtraining can be avoided. Usually a compromise solution is adopted such that the performance on either set is equal.

Finally, the network is presented with the validation set and its performance measured on this unseen data. It should be noted that this method requires many hidden units to avoid local minima, Sarle [92]. Further, Sarle notes that if there are at least twice as many training cases as there are weights in the network, there should be little overfitting. If the data is not noise free, then it should be thirty times as much, a point also alluded to by Haykin [43]. In fact, the data generated by the finite difference scheme will not be noise free since the data will contain truncation errors from the finite difference modelling. As will be seen later, this implies a river length (of data) about 1200 Km long, based on the spatial distance in the

finite difference scheme of Chapter 3, or alternatively, much smaller spatial distances and time steps. A simple example of early stopping is depicted in fig. 2.13:

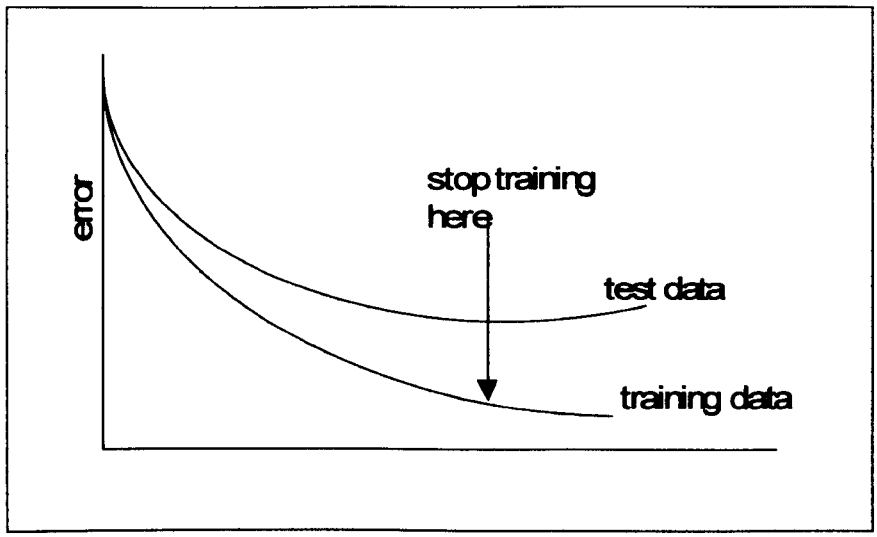

Figure 2.13: Early Stopping

2.5.9 Hold Out

In this alternative method, the training data is used to train several different architectures i.e. different numbers of hidden units in the hidden layer. The validation set is then used to select the best architecture based on a comparison of the error functions from the different networks. Finally, a test set is used to verify the final choice.

There is however, some confusion over the definition of training and validation sets. Some training methods such as 'early stopping' as mentioned above (cf. Section 2.5.8), require a validation set during this phase, though other methods do not. Ripley [89] and Bishop [9] have produced authorative and virtually similar definitions of these different sets of data. In essence, they state that:

- *Training Set* - used to minimize the error function on SEVERAL networks and hence tune parameters such as the weights.
- *Validation Set* - used to evaluate the performance of each of these several networks using an unseen data set. It is essentially used to choose the architecture such as the number of hidden units. The network with the smallest error is then selected. This method is called the 'hold out' method.

- *Test Set* – as this 'hold out' method could also lead to some overfitting to the validation set, the test set is used to measure the performance of the final selected network.

As to whether or not to have a validation set that is just that and not also used for training will depend on the amount of available data sets. In other words, is there sufficient data available to have a training set, validation set and test set or alternatively, just a training set and test set? Some researchers, Sarle [92], have noted that training a net with 20 hidden units requires anything from 150 to 2500 training cases.

The feed-forward network is good at generalization with regard to interpolation of related data but not at extrapolation (i.e. data that is not related to the training set). When the training data set is 'noise' free, that is the data is devoid of degradation or inaccuracies, sometimes the addition of noisy data can help convergence. Some suggested causes of overfitting by various researchers, Sarle [92], are:

- Too many weights or weights are too large
- Number of weights being far greater than the number of training patterns.

and some suggested means of avoiding it being:

- Use of 'regularization' or weight decay
- Use at least 30 times as many training cases as there are weights or 5 times as many if the data is noise free (as previously noted).

2.5.10 Supervised Learning

By far, the majority of networks are trained using supervised learning. In this mode of learning, both inputs and desired (target) outputs are provided to the network and after processing, the network compares the actual resultant outputs with the desired outputs. The network is able to evaluate these differences (errors) and adjust the initial weights accordingly so that on each iteration of processing, these differences are reduced.

When a desired accuracy, as determined by the user of the system, has been achieved, the training is considered to be complete. These idealized weights are then 'fixed' for use of the application on other data sets. It should be noted that training can be very time consuming

and a large number of data sets are needed for the training to be effective. Some networks never learn, the training suffering from various deficiencies such as 'local minima', 'overfitting', (as previously mentioned), poor 'convergence' and insufficient or unrelated data. Further, for multilayered networks with multiple nodes, there is the danger of 'associative memory' developing. In fact, after learning has taken place, it is possible to recall the complete original pattern even with incomplete or inexact input. It is important that the network has learned the general patterns in the data rather than memorizing the actual data itself.

Once the training is over, a test set of data that the network has not seen before is presented to it. The outputs from this when compared to the actual target outputs determine if the system needs further training. The network is not self-organizing and does not use a competitive learning rule method but instead uses supervised learning algorithms such as 'error-correction learning'.

2.5.11 Unsupervised Learning

Here only the inputs of a data set are provided to the network. The actual target outputs are not presented to it. The system looks for trends and regularities in the data and makes adaptions accordingly, the network being internally self-organizing. However, strictly speaking, it is only semi-automatic since a human must at some stage examine the results to determine when the training needs to stop. Weights and other parameters are adjusted once the output has been examined. A competitive learning rule algorithm is used wherein only the weights belonging to a 'winning' processing node will be updated. Other learning algorithms frequently used are 'reinforcement learning' and 'Hebbian Learning'. Well known examples of networks that use unsupervised learning methods are 'Adaptive Resonance Theory' (ART) and Kohonen's 'Self-Organizing Feature Map' (SOM).

2.6 Advantages and limitations of neural networks

The advantages and limitations of using neural networks can be summarized, by no means exhaustively, as follows:

2.6.1 Advantages

They are very useful in solving unstructured and semi-structured problems and excel at pattern recognition even from incomplete information, Turban and Aronson [102]. They are very adept at classification and abstraction, and are good at generalization, meaning they have the ability to produce reasonable outputs from inputs not encountered during training.

In contrast to the serial nature of processing in conventional computer systems, by the very nature of their construction, there are large numbers of interconnected neurons (processing units) all working on the same problem in parallel resulting in high speed processing.

A very useful quality of ANNs is their ability to adapt to changes in the environment. In other words, they can be easily retrained on new data. Further, they have the facility to cope with fault-tolerance situations. That is, even if a neuron or its surrounding connections are damaged, because of the distributed nature of information storage and processing, the performance is only slightly degraded. For the performance to be seriously impaired, the damage would need to be extensive. In fact, a neural network can also modify its own topology – thus emulating the restructuring of synaptic connections in the brain.

2.6.2 Limitations

Neural networks are not particularly good at performing tasks that involve complex numerical calculations and data processing. Such tasks are best addressed using conventional computer programs. In many respects, they are 'black box' scenarios in that there is an inherent lack of explanatory capability with no obvious interpretation of the connection weights and subsequent changes after training.

By their very nature, they need a vast amount of data for training and currently training (and hence retraining) times can be excessive and tedious making frequent retraining impractical. Most applications are restricted to software simulations as the cost of, and limitations of, current hardware render it not economically viable for 'hardwiring'.

2.7 A brief history of neural networks

The development of neural networks has been influenced by several fields of research namely, neuroscience, psychology as well as engineering.

The first neural network investigation was conducted by Alexander Bain (1873). Strictly speaking, however, this was a biological analysis of the neural network in the brain rather than an artificial one. As a result of Bain's work and that of William James (1890) et. al., the concept of a neuron was conceived between 1890 and 1910.

The earliest research into ANNs was in the 1940's when in that decade, physiologists McCulloch and Pitts (1943) developed the first artificial neural network based on their understanding of neurology. Their networks were based on simple neurons, which were considered to be simple logic functions with fixed thresholds. In other words, the Boolean functions AND and OR. Towards the end of the decade, Donald Hebb (1949) published some work based on simulations of a simple connected network. Based on the work of Hebb, Farley and Clark (1954) investigated simple pattern classification networks.

Following this, in the 1950's, the physiologist Rosenblatt [90] designed a two-layer network called 'the perceptron' wherein the weights were adjusted in a trial and error method. To improve upon this, Selfridge (1958) introduced the idea of gradient descent to the perceptron to adjust the weights. Although the perceptron was successful in classifying certain *patterns* (by adjusting the connection weights), it had its limitations. It could not for example, solve the classic XOR problem (Minksy and Papert, [79]) and such limitations led to a decline in the development of neural networks. Notwithstanding this, the perceptron was the cornerstone for later research into neural computing although ironically, the perceptron is not as good a model of the electrochemcial processes within the neuron (of the brain) as the model of McCulloch and Pitts (Anderson and Rosenfeld, [5]). Following similar lines of research to Hebb, Widrow and Hoff [109] also developed a learning rule, now often referred to as the *Delta learning rule*. This rule was later extended and became known as the Generalized Delta Rule (frequently referred to as backpropagation). The training process is similar to that of the delta rule for the simple perceptron. The

Generalized Delta Rule will be discussed further in Chapter 4 when the neural network model of the fluid flow is analysed.

In the 1970's, multi-layer neural networks were studied. Werbos [107] developed the back propagation learning method, in essence, a perceptron with multiple layers, different threshold function and a more robust learning rule. Further, other researchers showed renewed interest in neural networks in the early 1980's with the development of Boltzmann machines, Hopfield nets, competitive learning models, multilayer networks, and adaptive resonance theory (ART, Carpenter and Grossberg, [18]) models.

In the 1990's, neural networks became 'legitimate' as a computing tool and entered the field of mainstream applications. Since they are good at pattern recognition and are particularly useful in identifying patterns or trends in data, they are in use in such diverse applications as: OCR, speech recognition, stock market analysis, credit card approval, bankruptcy prediction, sales forecasting, customer research, data validation, risk management, target marketing, industrial process control and robotics. Further, the field of medicine is a particularly flourishing area of research for ANNs and of late, civil engineering, hydrology and hydrodynamics. In fact, of late, neural networks are now even being applied to climate modelling, Dibike and Coulibaly [32].

Chapter 3
The Hydraulic Model

3.1 Introduction

The prime difficulty with modelling river and estuarine flows is the lack of authentic measured data as it is extremely expensive to obtain this data by direct observation. Therefore, the main thrust of this chapter, after reviewing some previous research work, is to design a model that will generate this data (albeit in an artificial form) that can then be used to train and test artificial neural network models of the river flow. Thus, this chapter revolves around the development of a finite difference scheme based on the one dimensional St. Venant equations (shallow water theory), Dibike [31] and Vreugdenhil [106], from which a model without bottom friction and then a second model, with bottom friction, are developed. Although some non-linear terms such as advection are ignored, the shallow water equations represent a very useful and popular method of obtaining simplified models of river flow.

3.2 Previous Research

The mathematical modelling of river flow has been investigated by many researchers and where research has taken place into river hydrodynamics, it has often been far up stream well away from the estuarine areas. Some notable exceptions to this are Johns [50], Lin and Falconer [66], Kachroo [53], Kachroo and Liang [54], Kachroo et. al. [56], Kachroo et. al. [55] and Matalas [75]. All these researchers used 'classical' mathematical modelling techniques rather than neural networks. Lin and Falconer [66] developed a three-dimensional model using finite difference methods to predict sediment fluxes in the estuarine and coastal waters of the Humber River. Kachroo [53], Kachroo and Liang [54] investigated the modelling of river flow forecasting using methods such as linear perturbation theory with, in the latter paper, an analysis of multiple input / single output regimes using least squares techniques. Kachroo et. al. [56], Kachroo et. al. [55] further extended this investigation into the same problem with an emphasis on linear modelling

techniques. Bose and Jenkins [10], and Delleur et. al. [26] also applied classical methods (no neural network applications) to hydrographic time series models. The rest of the reviewed papers in this section have been grouped according to the area of application of neural networks.

3.2.1 Water resources and sediment transfer:

Bowden et. al. [12] applied neural networks to the modelling of water resources whilst Aitkenhead et. al. [4] applied them to environmental systems. Abrahart and White [1] and Cigizoglu [24] investigated the neural network modelling of sediment transfer in rivers and in a similar vein, Maier and Dandy [68] and Bowden et.al. [13] studied the transfer of salt in rivers. They [Maier and Dandy] applied neural network modelling to the forecasting of salinity in the River Murray near Adelaide, Australia to minimize the damage caused by salt intrusion. They obtained results that were rather encouraging with an absolute percentage error of just 6.8% for a 14-day forecast. They noted that this was somewhat better than the previously used time-series model.

Sandhu et. al. [91] applied analytical hydrodynamical modelling techniques in the Sacramento-San Joaquin Delta to obtain real time estimates of salinity and pollutant concentrations in the water. It was noted that the models had only limited success since being a delta, the flow was of a highly 'multiple input / single output' (MISO) nature. They later applied the techniques of ANNs to their models and calibrated them against salinity measurements taken at various points in the delta. They observed that much greater 'flexibility' was possible in the models in that they were not confined to very restricted pre-determined hydrodynamic methods. They were able to model more accurately multiple inputs (that is at a river junction) rather than single input flow and other flows from around physical obstructions. Furthermore, there was a much greater accuracy in terms of real time estimates of the salinity concentration.

ANNs have also been used by the Delft Research Institute in Holland, Schleider and Cser [93], and the Groundwater Research Centre of the Technical University of Denmark, Sonnenborg [98]. The research at Delft has concentrated mainly on hydraulics, beach

erosion and wind 'events' whilst the research in Denmark was aimed principally at solving the problem of saltwater intrusion into coastal freshwater supplies.

3.2.2 Offshore modelling:

Somewhat slightly divorced from the theme of river investigations, the next group of researchers concentrated on offshore modelling. Agrawal and Deo [3] applied neural networks to the problem of wave analysis and the interrelationships between certain characteristics of wave parameters. They investigated the usage of neural networks in order to capture the non-linearity (of the mathematical relationships between the parameters) in a matrix of weights and bias values during the learning process. The network was trained using observed data from an offshore site along the east coast of India.

El-Rabbany et. al. [35] considered the difficulties of tidal prediction using the traditional standard harmonic method of using existing tidal records. To overcome the inaccuracies in the data, a modular, three-layer, feedforward neural network trained using the back-propagation algorithm was developed. They found that the maximum prediction error in the tidal height was approximately 20 cm, whilst most of the residual errors fell within the +/-10 cm range. Comparing the results with a sequential least squares method for tidal prediction, it was noted that the neural network model demonstrated an improvement by a factor of five over the least squares method, even when long tidal records were unavailable.

Huang et. al. [46] discuss the application of a neural network to, as with El-Rabbany et. al. [35], the prediction of coastal water levels, in this case, at coastal inlets along the south shore of Long Island, New York. They were particularly interested in coastal hydrodynamics sediment transport. Due to the complex coastal and estuarine topography and shallow water effects, it was often difficult to obtain a good linear regression relationship between water levels from two different monitoring stations. They employed a three-layer, feed-forward, backpropagation structure, the training method being optimized using a conjugate training algorithm. They obtained very good long-term predictions of both non-tidal and tidal water levels at the regional coastal inlets.

Makarynskyy [69] investigated the use of artificial neural networks to predict wave heights offshore of the Irish coast. It was found that both the accuracy of the simulations and the

ability of neural networks to improve the initial forecasts (estimated in terms of the correlation coefficient, root mean square error and scatter index), depended on buoy location. In a further communication, Makarynskyy [70] provides much more detail on the work in [69]. Makarynskyy et. al. [71] extended the work of [69] to model wave predictions off the west coast of Portugal.

3.2.3 River/estuary modelling:

Atiya et. al. [7] studied river flow forecasting with two goals in mind. Firstly, they applied a neural network technique to the forecasting of river flows in the river Nile. Secondly, they used the time series data to benchmark different neural network model approaches to this river flow forecasting. The research involved simulating the time series (of flow data) using 4000 iterations and a neural network architecture with three hidden nodes. It was observed that varying the number of hidden nodes between two and eight made little difference to the results.

Chang and Chen [20] applied a radial basis function neural network to construct a water stage forecasting model for an estuary under high flood and tidal effects. Data from the Tanshui River in China was used for the training of the network.

Cigizoglu [22] considered the more unusual case of intermittent river flow i.e. wet (flow) and dry (no flow) periods. In his work, a neural network was used to forecast the daily intermittent river flows for a river in Turkey. Comparing the forecasted time series with the observed ones, it was noted that there was a good measure of agreement between the two with regard to the dry periods so that it would appear that the neural network was able to capture the transition between wet and dry periods. Further, the neural network approach was superior to classical regression in this particular study. Following on from his previous work [22], Cigizoglu [23], further investigated the applicability of neural networks to the forecast, estimation and extrapolation of the daily flow data of other rivers in the Eastern Mediterranean region of Turkey. The main objective was to compare a multi-layer perceptron model with that of a conventional statistical/stochastic model of the daily flow. He concluded that the superiority of the neural network over conventional methods could be

attributed to the ability of the network to capture the non-linear dynamics and generalise the structure of the whole data set.

Dibike and Solomatine [30] studied the applicability of neural networks for downstream flow forecasting in the Apure river basin, Venezuela. Using both a multi-layer perceptron and the radial basis function, they found that the results for river flow forecasting were slightly better than those obtained using a conceptual rainfall-runoff model. Rainfall-runoff models are described fully in Chadwick and Morfett [19]. Dolling and Varas [33] investigated streamflow prediction using artificial neural networks. They further considered in the modelling, aspects such as input variable selection, model architecture and the learning process. Their work has been mentioned in particular because of the succinctness and clarity of their paper.

Imrie, Duncan and Korre [47] investigated the limitations of a neural network that had been trained on a dataset containing a limited range of values. They applied a modification of the cascade-correlation learning architecture during training to improve the network's generalization capabilities. Using data from three tributaries of the river Trent, they observed that the cascade-correlation algorithm was easier to use and produced better ANN models than those obtained using the standard backpropagation algorithm. Further, they noted that a function with a cubic polynomial format in the output layer might be necessary for ANNs to capture extreme values. In addition, since the transfer functions they applied at the hidden layer were bounded, the limits of these values (propagated from the hidden layer to the output layer) must be largely determined by the scaling of the input data.

Jain and Kumar [49] presented a de-trended and de-seasonalised time series dataset of the river flow in the Colorado River to a (three layer) feed forward multi-layer perceptron neural network. From their work, they concluded that the neural networks were able to capture the relationships among the historical flow data and the future flow data much better than some conventional time series models.

Karunanithi et. al. [57] investigated the application of a neural network as a predictor for flow prediction of the Huron River, Michigan. They addressed issues such as neural

network architecture selection, correct training algorithm as well data input, by using the cascade correlation algorithm.

Kerh and Lee [58] studied the forecasting of flood discharge upstream of the Kaoping River in Taiwan. Using a 21-15-1 back propagation neural network model, they observed that it performed relatively better than a conventional Muskingum method. The Muskingum method is fully described in Chadwick and Morfett [19]. The evaluation criteria used were: RMSE indices; efficiency coefficients; peak discharge errors; and peak time errors. The dominant factors affecting the accuracy were flood discharge and water stage (water level above a given datum).

Kisi [59] studied the potential of using a neural network for river flow modelling of the Goksudere River in Turkey. Using a three-layer back propagation neural network trained using a gradient descent algorithm, he was able to compare predicted monthly flows with a more conventional method. Based on the results, he noted that in general, the neural network predictions were better than those obtained using conventional approaches.

Kocjancic and Zupan [60] completed a very rigorous study of the influence of training set selection as well as the modelling technique used. Using conventional regression methods such as multiple linear regression and partial least squares regression as well as a feed-forward neural network (with the error back-propagation and Levenberg–Marquardt learning algorithm), they generated 125 regression models.

Kumar et. al. [61] compared the application of two different neural networks, namely a feed forward neural network and a recurrent neural network, to the prediction of river flow for the River Hemavathi in India. The recurrent neural network was trained using the method of ordered partial derivatives whilst the feedforward network was trained using the conventional backpropagation algorithm. They noted that the recurrent network outperformed the feedforward one and further, the architecture size and training time were smaller.

Moradkhani et. al. [80] compared a radial basis function with other algorithms and differing neural architectures for the forecasting of daily streamflow in the Salt River, Colorado. They noted the absence of any authoritative procedure to suggest how to partition a data set and

hence used cross-validation, which has the benefit of reducing the danger of overfitting. Riad et. al. [88] applied a neural network based model to predict river flow for the Ourika River basin using a multi-layered perceptron trained using the back propagation algorithm. Sivakumar et. al. [95] completed a study of river flow forecasts of the Chao Phraya River in Thailand. They concluded that a better type of ANN architecture other than the MLP would be more suitable for longer lead-time forecasts, the selection of the training set was crucial and also the influence of noise on the forecast accuracy needed to be further studied.

Thirumalaiah and Deo [101] studied the application of neural networks to real-time forecasting using three different algorithms. They used the backpropagation, cascade correlation, and conjugate gradient methods. They observed that the cascade correlation algorithm took only a small fraction of time to train the network in comparison to the backpropagation and conjugate gradient methods.

Zealand et. al. [111] applied ANNs to the simulation of short term forecasting of streamflow comparing also this methodology to that of more conventional approaches. They studied issues such as the type and size of the input data, sizes of the hidden layers and capabilities of the neural network to model complex non-linear relationships. Applying the model to the Winnipeg River system in Northern Ontario, Canada, their model outperformed a conventional Winnipeg Flow Forecasting System (WIFFS) model during the verification (testing) phase. It was noted that the greatest difficulty was in determining the appropriate model inputs and so they concluded that it was important to determine the dominant model inputs in order to reduce the size of the network and training time and improve its generalization ability. They achieved RMSE accuracies of 32.5 cms. Nayak et. al. [83] used fuzzy neural network techniques for river hydrograph analysis whilst Tawfik [100] concentrated on the applications of neural networks to river modelling of the River Nile.

Izquierdo et. al. [48] compared different models used in the water industry. Their research concentrated on comparisons between numerical, statistical and neural network techniques as well as fuzzy network methodologies.

3.2.4 Theoretical hydrodynamic application of neural networks:

Dibike and Abbott [27] studied the simulation of two-dimensional flows whilst Dibike and Minns [29] considered the application of neural networks to the generation of wave equations. Following on from the earlier work of Dibike et. al. [28] and Dibike and Solamatine [30], Dibike [31] studied the possibility, in the case of simple problems i.e. one and two dimensional flows, as to whether a generic neural network could 'construct itself' by learning from existing numerical hydraulic models. Applying the study to various types of bathymetries and time steps, he concluded that the neural network provided acceptable results and further, a well-trained network could even replace the finite difference schemes frequently used in numerical hydrodynamic modelling. However, most importantly, the effects of bottom friction had not been included. It should be noted that as he used linear activation functions (cf. Chapter 2), it would not have been possible to represent the non-linear effects of this bottom stress.

3.2.5 Hydrodynamic support vector machine applications:

Asefa et. al. [6] applied the methodology of support vector machines (SVMs) to multi-time scale predictions of stream flow in the Sevier River basin, near the Great Salt Lake. They noted the ability of SVMs to generalize, given small data sets and observed that Transfer Function Noise (TFN) models gave a smoother prediction, specifically for annual flow volumes, whilst the SVMs gave overall better Root Mean Square (RMSE) error values.

Han and Yang [42] studied the application of support vector machines to river flow modelling. In their paper, they tried to demonstrate the potential of support vector machines to dendritic river modelling. They noted that the empirical performance of support vector machines is generally as good as the best artificial neural network solutions.

Han et. al. [40] investigated flood forecasting using support vector machines in the Bird Creek catchment area at Owasso in the USA. They observed that SVMs like artificial neural networks, also suffered from over-fitting and under-fitting but further, the selection of various input combinations and parameters was very problematic. Notwithstanding this, they obtained quite credible performance from the SVMs.

3.3 Shallow water theory

Numerical models based on non-linear shallow water theory are potentially very efficient in terms of speed of simulation, the type of wave being classified according to the relative depth ratio H/L, where H is the water depth and L the inshore wavelength as follows:

Ratio of H/L	**Type of wave**
H/L <= 0.05	Shallow Water Waves (Long Waves)
0.05 < H/L < 0.5	Intermediate Depth Waves
H/L > 0.5	Deep Water Waves (Short Waves)

Table 3.1: Wave Classification According to Relative Depth

In the area of interest in this research, the lower reaches of the Thames at its confluence with the river Medway, the average depth of water is approximately 15 m. The wavelength is equal to the tidal period Tp times the phase speed (celerity) c, where the tidal period is 12.4 hours and c is equal to $\sqrt{gH}$. Hence,

$$L = T_p \times c = 12.4 \times 3600 \times \sqrt{9.81 \times 15} = 541.51 Km.$$

Thus $\frac{H}{L} = \frac{15}{541510} = 0.000028$ approximately and so a shallow water equation system is valid in the area of interest.

The non-linear hyperbolic shallow water equations can be derived from the depth-averaged Navier-Stokes equations of hydrodynamics, Ponce and Simons [86], wherein the conservation of mass and momentum of fluids are represented. In non-linear shallow water theory, stresses such as wind shear are ignored, the main assumptions adopted in the derivation of these (shallow water) equations being:

- the vertical velocity is small in comparison to horizontal velocity;
- the pressure distribution in the vertical is hydrostatic.

Further, the one dimensional variant of these shallow water equations (the Saint-Venant system), Gerbeau and Perthame [36], Ponce and Simons [86], originally developed to model near horizontal, free-surface channel flows such as rivers and near coastal regimes, assumes

a small bed slope that usually incorporates a simplified bed shear stress term. Sometimes an extra system equation is added to model the transport of pollutant (or temperature).

3.4 A simple hydrodynamic model

The primitive equations of de Saint Venant (1850), Dibike [31], for a one-dimensional nearly horizontal free surface flow (in Eulerian form) can be written:-

Continuity (mass conservation):

$$\frac{\partial H}{\partial t} + H\frac{\partial u}{\partial x} + u\frac{\partial H}{\partial x} = 0 \tag{3.1}$$

Momentum:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + g\frac{\partial H}{\partial x} = 0 \tag{3.2}$$

where t and x represent time and distance (along the river) respectively, g is the gravitational acceleration, H is the water depth of the channel and u is the depth-averaged water velocity. It is assumed that the slope of the channel bottom is constant and horizontal.

If the waves are very long with amplitudes that are relatively small compared to the water depth, the advection terms

$$u\frac{\partial u}{\partial x} \quad \text{and} \quad u\frac{\partial H}{\partial x}$$

in **(3.1)** and **(3.2)** are of a lower order than the other terms. This is certainly true of the area of interest in the Thames/Medway confluence where the channel depth is of the order of 15m so that the velocity u is much less than $\sqrt{gH}$. Consequently, these two equations can be simplified to a first order linearized approximation as:

$$\frac{\partial H}{\partial t} + H\frac{\partial u}{\partial x} = 0 \tag{3.3}$$

$$\frac{\partial u}{\partial t} + g\frac{\partial H}{\partial x} = 0 \tag{3.4}$$

Following the arguments of Vreugdenhil [106], set $H = h + \xi$ where h is the still water depth and $\xi \ll h$. The latter condition is tantamount to saying $u \ll \sqrt{gH}$ as previously noted. Then (3.3) and (3.4) become

$$\frac{\partial \xi}{\partial t} + h\frac{\partial u}{\partial x} = 0 \tag{3.5}$$

$$\frac{\partial u}{\partial t} + g\frac{\partial \xi}{\partial x} = 0 \tag{3.6}$$

where ξ is the amplitude of the wave height as can be seen in fig. 3.1:

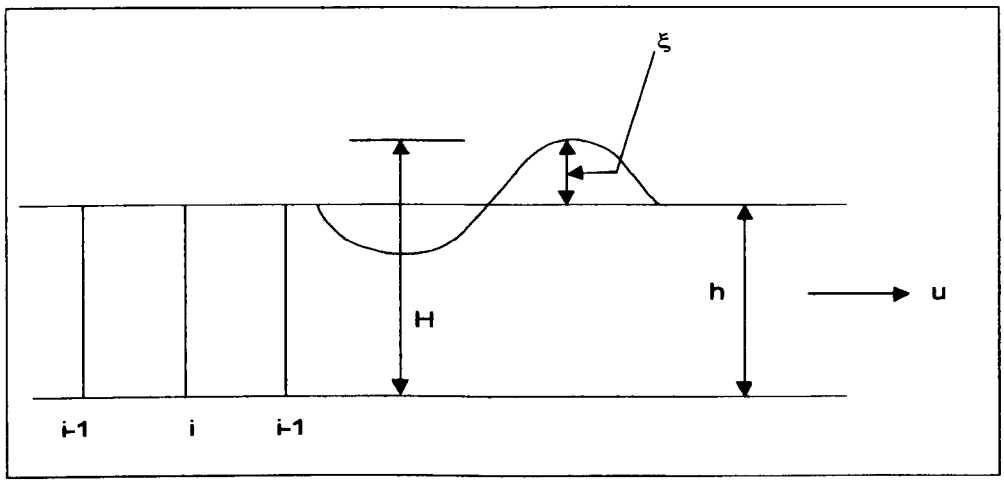

Figure 3.1: Fluctuations above/below still water level

3.5 Finite Difference Scheme – zero bottom friction model

A two level time scheme as shown in fig. 3.2 is used

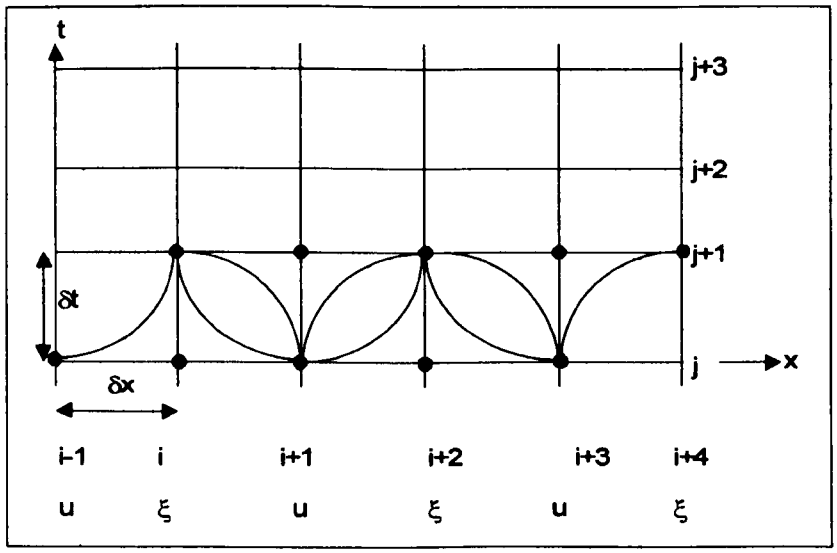

Figure 3.2: Leapfrog Mesh

since knowledge of the system is only available at one time level, at time t = 0. A mesh system that is staggered in space and time, and commonly referred to as a *leapfrog* scheme, is used to develop a finite difference scheme for equations **(3.5)** and **(3.6)** as follows:

From equations **(B12)** to **(B15)** appendix B (finite difference notes), using a forward difference scheme for time and a central difference scheme for the distance along the river i.e. x, where the mesh size is now $2\delta x$, the continuity and momentum equations **(3.5)** and **(3.6)** can be represented by the following explicit difference scheme

$$\frac{\xi_i^{j+1} - \xi_i^j}{\delta t} + h\frac{u_{i+1}^j - u_{i-1}^j}{2\delta x} = 0 \qquad \textbf{(3.7)}$$

$$\frac{u_{i+1}^{j+1} - u_{i+1}^j}{\delta t} + g\frac{\xi_{i+2}^{j+1} - \xi_i^{j+1}}{2\delta x} = 0 \qquad \textbf{(3.8)}$$

As can be seen from fig. 3.2, initial values for t and u need only be known at one time level. Since there is no *a priori* knowledge of the flow other than the imposed flows at the boundaries and assumed zero values for the velocities at time zero, only a two level time scheme such as that in the fig. 3.2 could be used. It is now necessary to rearrange these two formulae into a suitable format for computation. For an *explicit* hydrodynamic scheme to be stable, it is required that the Courant number, denoted by Cr and which is dimensionless, to be <=1. Here $Cr = c\frac{\delta t}{\delta x}$ where c, the wave celerity (or phase speed), is, as previously noted, defined by $c^2 = gh$. In fact, with $g = 9.81$, $h = 15$ m, the only requirement therefore is that

$$\frac{\delta t}{\delta x} \le \frac{1}{c} = \frac{1}{\sqrt{9.81 \times 15}} = 0.082437.$$

A reasonable choice for the spatial interval was δx = 500 m which would provide 60 velocity and 60 depth sections over a 60 Km-length of river. Further, this was the spatial interval used in a numerical model of Johns [51] (see Chapter 4) used to verify some of the work in this research. This choice of spatial interval implies that

$$\delta t \le 0.082437\delta x = 41.2185 \text{ secs.}$$

Although therefore in principle the temporal interval could be quite small, 5 secs. for instance but which would incur many time levels and hence long simulation times, it was decided to adopt δt = 30 secs. This value thus results in Cr = 0.7278.

After a little rearrangement, **(3.7)** becomes

$$\begin{aligned}\xi_i^{j+1} &= \xi_i^j - h\frac{\delta t}{2\delta x}\left(u_{i+1}^j - u_{i-1}^j\right)\\ &= \xi_i^j - \frac{c^2}{g}\frac{\delta t}{2\delta x}\left(u_{i+1}^j - u_{i-1}^j\right)\\ &= \xi_i^j - \frac{Cr}{2}\sqrt{\frac{h}{g}}\left(u_{i+1}^j - u_{i-1}^j\right)\end{aligned} \qquad \textbf{(3.9)}$$

and **(3.8)** transposes to

$$\begin{aligned}u_{i+1}^{j+1} &= u_{i+1}^j - g\frac{\delta t}{2\delta x}\left(\xi_{i+2}^{j+1} - \xi_i^{j+1}\right)\\ &= u_{i+1}^j - \frac{c^2}{h}\frac{\delta t}{2\delta x}\left(\xi_{i+2}^{j+1} - \xi_i^{j+1}\right)\\ &= u_{i+1}^j - \frac{Cr}{2}\sqrt{\frac{g}{h}}\left(\xi_{i+2}^{j+1} - \xi_i^{j+1}\right)\end{aligned} \qquad \textbf{(3.10)}$$

These final forms i.e. **(3.9)** and **(3.10)** are the finite difference schemes used in the computations for the cases where there is no bed friction. To generate the test data, ξ has to be evaluated using **(3.9)** at all alternate grid points at time level *(j+1)*. Then, *u* has to be similarly evaluated using **(3.10)** at the other alternate grid points at time level *(j+1)*. Once complete, the process is then repeated at time level *(j+2)* and so on.

It should be noted that in a similar development, Dibike [31] has incorrectly formulated the coefficients in **(3.9)** and **(3.10)** above. He had $\sqrt{\frac{g}{h}}$ instead of $\sqrt{\frac{h}{g}}$ and vice versa which was obviously a typographical error.

3.5.1 Stability of the Finite Difference Scheme

The stability of a finite difference scheme depends on whether or not errors are magnified as computation proceeds up the time levels. Provided these errors are not magnified but bounded, the scheme can be regarded as stable. The analysis of such stability relies on the use of the Von Neumann method, Vreugdenhill [106], wherein the errors can be represented in the form of a Fourier series. In essence that is, the variables ξ and u can be represented in terms of harmonic functions by setting $\xi_i^j \equiv H^j e^{ki\lambda}$ and $u_i^j \equiv U^j e^{ki\lambda}$ where $k = \sqrt{-1}$ and λ is some arbitrary variable. Note that H^j and U^j are the height and velocity amplitudes at time level 'j'. Hence rearranging **(3.9)** and **(3.10)** and making these substitutions gives

$$H^{j+1}e^{ki\lambda} - H^{j}e^{ki\lambda} + \frac{Cr}{2}\sqrt{\frac{h}{g}}\left(U^{j}e^{k(i+1)\lambda} - U^{j}e^{k(i-1)\lambda}\right) = 0 \tag{3.11}$$

$$U^{j+1}e^{k(i+1)\lambda} - U^{j}e^{k(i+1)\lambda} + \frac{Cr}{2}\sqrt{\frac{g}{h}}\left(H^{j+1}e^{k(i+2)\lambda} - H^{j+1}e^{ki\lambda}\right) = 0 \tag{3.12}$$

After a little manipulation, **(3.11)** and **(3.12)** simplify respectively to

$$H^{j+1} - H^{j} + Cr\sqrt{\frac{h}{g}}U^{j}k\sin\lambda = 0 \tag{3.13}$$

and

$$U^{j+1} - U^{j} + Cr\sqrt{\frac{g}{h}}H^{j+1}k\sin\lambda = 0 \tag{3.14}$$

The characterisation of the variables in terms of harmonic functions enables the application of the further substitutions $U^{j} = \rho^{j}U$ and $H^{j} = \rho^{j}H$. The variable ρ represents the so called amplification factor and is a measure of how the errors are amplified from one time level to the next during the calculations. Ideally, the requirement is that $|\rho| \le 1$. Here U and H are the amplitudes at the initial time level. Making these substitutions, **(3.13)** and **(3.14)** become

$$\rho^{j+1}H - \rho^{j}H + Cr\sqrt{\frac{h}{g}}\rho^{j}Uk\sin\lambda = 0$$

and

$$\rho^{j+1}U - \rho^{j}U + Cr\sqrt{\frac{g}{h}}\rho^{j+1}Hk\sin\lambda = 0$$

which upon division by ρ^{j} ,simplify even further still to

$$\rho H - H + Cr\sqrt{\frac{h}{g}}Uk\sin\lambda = 0 \tag{3.15}$$

and

$$\rho U - U + Cr\sqrt{\frac{g}{h}}\rho Hk\sin\lambda = 0 \tag{3.16}$$

The last two equations **(3.15)** and **(3.16)** are a pair of linear homogeneous equations that can be written in matrix form as:

$$\begin{pmatrix} \rho-1 & Cr\sqrt{\frac{h}{g}}k\sin\lambda \\ Cr\sqrt{\frac{g}{h}}\rho k\sin\lambda & \rho-1 \end{pmatrix}\begin{pmatrix} H \\ U \end{pmatrix}=\begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{3.17}$$

which has a non-zero solution only if the determinant of the coefficients is zero. Bearing in mind that having previously defined k as the imaginary number $\sqrt{-1}$, the determinant of **(3.17)** gives

$$(\rho-1)^2+(Cr\sin\lambda)^2\rho=0 \tag{3.18}$$

which upon expansion results in a quadratic in ρ with two roots, one real part and the other a complex one. The real root represents a physical wave whilst the complex one is of a purely numerical origin that would give rise to a wave of an oscillatory character and as such needs to be bounded. Analysis of the complex conjugate part results in the boundness condition:

$$\left(\frac{(Cr\sin\lambda)^2}{2}-1\right)^2-1\le 0 \tag{3.19}$$

Since this condition must hold for even the most unfavourable value for λ, $\lambda=\frac{\pi^r}{2}$ say, then $|\rho|^2\equiv\rho\bar{\rho}=1$ provided $|Cr|\le 2$. The obtained value for Cr = 0.7278, (cf. Section 3.5), obviously therefore satisfies this requirement.

3.5.2 Compatibility of the Finite Difference Scheme

The various terms in equations **(3.9)** and **(3.10)** can be expanded as follows in a Taylor's series:

$$\xi_i^j\equiv\xi(x,t),$$

$$\xi_{i+1}^j\equiv\xi(x+\delta x,t)=\xi(x,t)+\delta x\frac{\partial\xi}{\partial x}(x,t)+\frac{(\delta x)^2}{2!}\frac{\partial^2\xi}{\partial x^2}(x,t)+O((\partial x)^3),$$

$$\xi_i^{j+1}\equiv\xi(x,t+\delta t)=\xi(x,t)+\delta t\frac{\partial\xi}{\partial t}(x,t)+\frac{(\delta t)^2}{2!}\frac{\partial^2\xi}{\partial t^2}(x,t)+O((\partial t)^3),$$

$$\xi_{i+2}^j\equiv\xi(x+2\delta x,t)=\xi(x,t)+2\delta x\frac{\partial\xi}{\partial x}(x+2\delta x,t)+\frac{2(\delta x)^2}{2!}\frac{\partial^2\xi}{\partial x^2}(x+2\delta x,t)+O((\delta x)^3)$$

$$\xi_{i+2}^{j+1} \equiv \xi(x+2\delta x, t+\delta t) = \xi(x,t) + 2\delta x \frac{\partial \xi}{\partial x}(x+2\delta x, t+\delta t) + \frac{2(\delta x)^2}{2!}\frac{\partial^2 \xi}{\partial x^2}(x+2\delta x, t+\delta t) + O((\delta x)^3) + \delta t \frac{\partial \xi}{\partial t}(x+2\delta x, t+\delta t) + \frac{(\delta t)^2}{2!}\frac{\partial^2 \xi}{\partial t^2}(x+2\delta x, t+\delta t) + O((\delta t)^3)$$

$$u_i^j \equiv u(x,t),$$

$$u_{i-1}^j \equiv u(x-\delta x, t) = u(x,t) - \delta x \frac{\partial u}{\partial x}(x,t) + \frac{(\delta x)^2}{2!}\frac{\partial^2 u}{\partial x^2}(x,t) + O((\partial x)^3),$$

$$u_{i+1}^j \equiv u(x+\delta x, t) = u(x,t) + \delta x \frac{\partial u}{\partial x}(x,t) + \frac{(\delta x)^2}{2!}\frac{\partial^2 u}{\partial x^2}(x,t) + O((\partial x)^3),$$

$$u_{i+1}^{j+1} \equiv u(x+\delta x, t+\delta t) = u(x,t) + \delta x \frac{\partial u}{\partial x}(x+\delta x, t+\delta t) + \frac{(\delta x)^2}{2!}\frac{\partial^2 u}{\partial x^2}(x+\delta x, t+\delta t) + O((\delta x)^3) + \delta t \frac{\partial u}{\partial t}(x+\delta x, t+\delta t) + \frac{(\delta t)^2}{2!}\frac{\partial^2 u}{\partial t^2}(x+\delta x, t+\delta t) + O((\delta t)^3).$$

Inserting these expansions into **(3.9)** and **(3.10)** and subtracting the original differential equations **(3.5)** and **(3.6),** it is possible to obtain the truncation errors for both difference equations. The truncation error *'T'* in both cases was of first order in both δt and δx, that is:

$$T = O(\delta t) + O(\delta x)$$

This is as it should be since the original differential equations, that is equations **(3.5)** and **(3.6)** contained only first order partial derivatives. The finite difference equations, equations **(3.7)** and **(3.8)** are therefore 'compatible' with the original differential equations **(3.5)** and **(3.6)**. This compatibility is an absolute requirement. Further, as a result of this compatibility, Smith [97], and the analysis earlier on the stability and amplification criteria, the implication is that as the mesh size $\delta t, \delta x$ gets smaller, then the finite difference scheme should approach (converge to) the true solution of the differential equations themselves.

Vreugdenhill [106] had performed a similar analysis for a three level time scheme using centred differences in ***both*** time and spatial intervals and as a result obtained $|Cr| \leq 1$ It is usual in practice to use this latter bound and this has been adopted in this research, the

actual value of Cr being determined, subject to these constraints, by the choice of 'sensible' values for δt and δx as previously discussed earlier. Classical theory indicates that explicit finite difference schemes are only conditionally stable if $Cr <= 1$. This, however is for 'non-mixed' schemes, that is where there is only one variable such as ξ or h. However, the difference scheme in this research involves the use of two variables ξ and h in both equations **(3.5)** and **(3.6).**

3.6 Boundary Conditions

The adopted convention is that U_o represents the imposed velocity at the upstream end of the river and U_L the (forcing tidal) velocity at the downstream (estuarine) end of the river. Writing the tidal input this way is more convenient than using the wave amplitude. However, since the amplitude is sinusoidal, a positive or negative value will 'flip' the direction by 180^o and cause the tidal wave to reverse direction. A further adopted convention is that the velocity is positive when flowing from the upstream to the downstream end. The initial value of U_o is assumed constant and positive so that it represents normal input into a river at source or a gauge station.

It would appear at first sight that the model adheres to the 'law of the wall'. The latter refers to the representation of the velocity profile in the turbulent boundary layer adjacent to the floor of the river. Gerbeau and Perthame [36] developed a 'Saint-Venant system with friction' by maintaining the viscous terms in the original Navier-Stokes equations. Their model therefore includes friction effects (not to be confused with bed friction – see Section 3.12) and as such obeys the 'law of the wall'. If there is a region close to the bed where viscosity is important and that the bottom shear stress (due to viscosity) is the important constraint on the flow, the following functional relationship for the velocity distribution may be obtained:

$$\frac{u}{u_\tau} = f\left(y\frac{u_\tau}{\nu}\right)$$

where y is the distance from the bottom, ν is the kinematic viscosity and u_τ is the shear stress (frictional velocity in the boundary layer). That is, if $y\frac{u_\tau}{\nu}$ is plotted against $\frac{u}{u_\tau}$ for many different flows, there will be a *single* curve. This is tantamount to saying that the

average velocity of a turbulent fluid at a point is proportional to the logarithm of the distance from the point to the bottom. However, unlike the model of Gerbeau and Perthame [36], the model as described by equations **(3.3)** and **(3.4)** does not contain the effect of viscosity as this was eliminated in the course of a first order linearization. That is, apart from 'bottom friction' (see Section 3.12), the model investigated in this research is assumed to be strictly frictionless and so does *not* obey the 'law of the wall'.

U_L can be entered as positive or negative during data entry in the simulations, the type of resultant flow being dependant on whether the first half (e.g. 25 hours) or second half (e.g. 25.5 hours) of a cycle has been selected. The advantage of being able to specify an ebb tide immediately is that the simulation does not need to cycle all the way through the first half of a cycle, the flood tide, to get to an ebb tide. The relationship is as follows:

Entered value of U_L	First half of a cycle e.g. 25, 25.125, 25.25, 25.375	Second half of a cycle e.g. 25.5, 25.625, 25.75, 25.875
U_L = positive	Flood	Ebb
U_L = negative	Ebb	Flood

Table 3.2: Flood and Ebb Tidal Cycles

The smallest fraction of a tidal cycle that can be selected is 0.0625.

This restriction on the fraction size is a consequence of the Matlab code (see VenantDataGenerator2 in Appendix C). The number of time steps in a tidal cycle with a temporal interval of 30 secs. is:

$$\frac{T_p}{\delta t} = \frac{12.4 \times 3600}{30} = 1488$$

Within the code, the range of the loop counters must be integers. Consequently selecting 25.1 tidal cycles for example would give:

$$25.1 \times 1488 = 37348.8$$

which is clearly unsatisfactory as it is a non-integer whilst

$$25.0625 \times 1488 = 37293$$

satisfies the integer requirement. With the tidal cycle T_P = 12.4 hours, 0.0625 equates to 46.5 minutes. It should be noted that this fraction size depends on the size of the temporal interval.

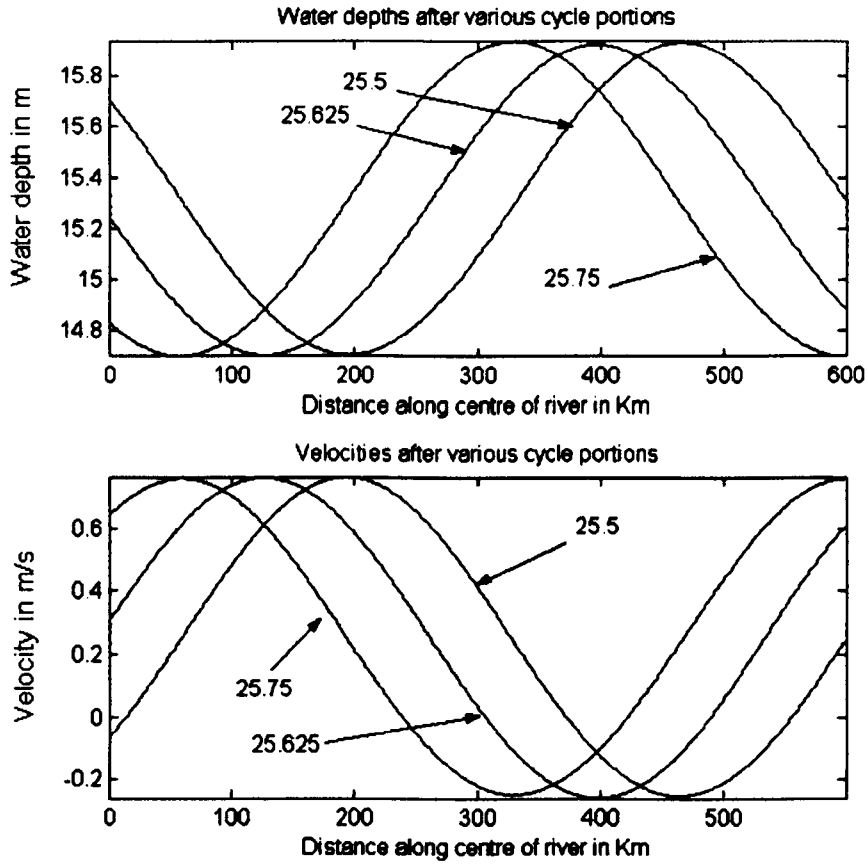

Figure 3.3: Flood and Ebb Tidal Cycles

From fig. 3.3 it can be seen that at 0 Km the water depth is about 15.7 m at tidal cycle 25.5 hours. Halfway through this ebb cycle, that is at 25.75 hours, the water depth has dropped to about 14.82 m. Similarly, at 0 Km the velocity is around -0.05 m/s indicating the flow is upstream even though this is an ebb phase. This is not unusual since there is always a little delay before the velocity approaches zero and then switches direction. Finally, the velocity is about 0.63 m/s halfway through the ebb phase at 25.75 hours. Since it is positive, it is in the direction of the estuary and so entirely consistent with an ebb flow scenario.

To use the model as given by **(3.9)** and **(3.10)** above would lead to spurious wave reflections from both the upstream boundary at x = 0 as well as a similar reflection at the downstream boundary at x = L. To allow water to flow freely either way through these boundaries

requires suitable boundary conditions based on the Theory of Characteristics e.g. Smith [97]. In a private communication with Johns [50], suitable boundary conditions based on characteristics were formulated.

At the upstream boundary x = 0, the condition is

$$\xi = \sqrt{\frac{h}{g}}\left(2u_0 - \overline{u_o}\right) \tag{3.20}$$

whilst at the downstream boundary x = L, it is

$$\xi = \sqrt{\frac{h}{g}}\left(2u_L \sin\left(\frac{2\pi t}{T_p}\right) + \overline{u_L}\right) \tag{3.21}$$

where u_0, u_L are the imposed velocities at the upstream and downstream boundaries respectively, $\overline{u_0}, \overline{u_L}$ are the depth averaged velocities at (or near to) these boundaries and T_p represents a typical tidal period in seconds. The tidal period in the case of the Thames is 12.4 hours as noted earlier.

The finite difference forms for the boundary conditions are then respectively:

$$\xi_1^{j+1} = \sqrt{\frac{h}{g}}\left(2u_o - \overline{u_o}^{j}\right) \tag{3.22}$$

$$\xi_L^{j+1} = \sqrt{\frac{h}{g}}\left(2u_L \sin\left(\frac{2\pi(j-1)\delta t}{T_p}\right) + \overline{u_L}^{j}\right) \tag{3.23}$$

It is noted that the values of ξ at the boundaries have to be computed first at each time level. The domain of analysis to be initially considered is 600 Km in length. In this way, Vreugdenhil [106], the boundaries can if necessary be moved well away from any particular area of interest within this domain.

3.7 Data generation for zero tidal flow

The Matlab (filename:VenantDataGenerator2) code used to generate the test data and subsequent plots is in appendix C. The program can generate data with or without the effects of bottom stress. The pseudo code is as follows:

Filename: VenantDataGenerator2

COMMENT: *The initial conditions are that velocity = 0 for all x and t = 0 except at x = 0 where there must be some basic input to start the system going. This is equivalent to a hydrographic input (boundary condition) at the river end of the estuary.*

COMMENT: *This program computes using FDS, the height above still water level and associated velocities, at the centre of the river given this boundary hydrographic velocity input at the river end of the estuary and a value for the average depth (still water level) of the channel. Since a staggered grid is being used, the spatial interval is 2 x deltax. Velocities are stored in U(j,n) and the height values are stored in H(j,n). Using two separate matrices like this means that each matrix will have alternate rows containing nothing other than zeroes (even after computing) as a consequence of the finite difference scheme staggered method.*

COMMENT: *The j index represents the rows of velocity and height respectively whilst the n index represents the time iteration levels.*

COMMENT: *There are velocity and depth estimates (sections) obtained using a given number of time iterations. The number of tidal cycles (for time iteration) and upstream and downstream velocities are selectable. This model covers a length of (sections-1)/2 km along the river.*

COMMENT: *Note, for stability of the FDS scheme, the time spatial grid has been chosen so that the Courant number is less than one.*

COMMENT: *Suitable boundary conditions as suggested by B.Johns are used to avoid wave reflection.*

PROCESS_*initialize data*

    INITIALIZE *space, time tidal cycle variables*

    GET *model length, velocity boundary conditions, still water depth*

    INITIALIZE *velocity and depth arrays*

    INITIALIZE *parameters g, c, Cr, k, m, l, p, r, q*

END

PROCESS_*calculate depths and velocities*

    LOOP_*on tidalcycles*

        LOOP_*on timesteps*

            calculate depths at upstream and downstream boundaries

```
                    LOOP_on sections
                        calculate depths at odd numbered sections
                    END
                    LOOP_on sections
                        calculate velocities on even numbered sections
                    END
                END
                IF no. of tidal cycles has not been surpassed
                    LOOP_on odd sections
                        transfer latest depth iteration to previous timestep
                    END
                    LOOP_on even sections
                        transfer latest velocity iteration to previous timestep
                    END
                END
            END
            LOOP_on sections
                convert velocities to depth averaged velocities
            END
        END
        PROCESS_plot depth graphs
            calculate average depth
            plot average depth
            plot depth
        END
        PROCESS_plot velocity graphs
            calculate average velocity
            plot average velocity
            plot velocity
        END
        PROCESS_save data
            save velocities and depths to a file
        END
```

All of the simulations to generate the data were run over a 25 tidal cycle period which is approximately equivalent to 12.92 days of real tidal flow in the Thames. As noted earlier, the speed of the disturbance is $\sqrt{gH}$ so that any disturbance flowing from the upstream end of the river can reach the 50 Km point of interest in approximately 1.14 hours which is about 0.09 tidal periods (T_p). Hence, a simulation over 10 tidal cycles would have been more than adequate. However, to see an extended area of interest as depicted later in figs. 3.11, 3.12 and 3.13, the disturbance would require 13.7 hours (or approximately 1.1 tidal cycles) to cover a distance of 600 Km and so a 120 tidal cycle period was chosen for the simulation shown in that figure. Three different scenarios of water depths were chosen namely:10 m, 15 m and 20 m. Further, at each of these depths, different configurations were chosen for the imposed inflow at the river end (such as would be represented by a hydrograph) and that at the estuarine boundary which would represent an imposed tidal oscillation. The depths were computed at sections along the river at 0, 2, 4...50 Km and velocities at 1, 3, 5...49 Km since the spatial interval is 2 x 500 m. Graphs of the water depths and velocities at the 15 m scenario for different configurations follow.

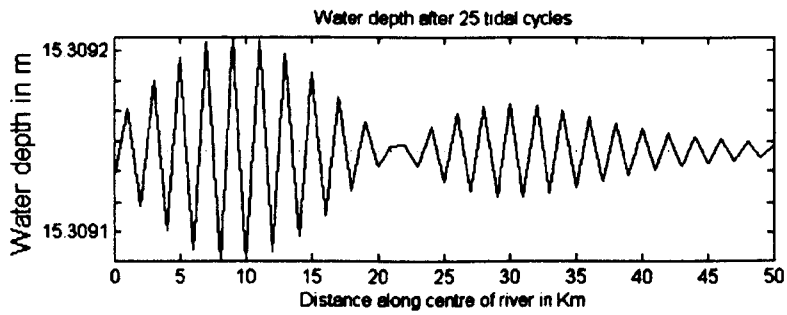

Figure 3.4a: Depth under zero tidal flow

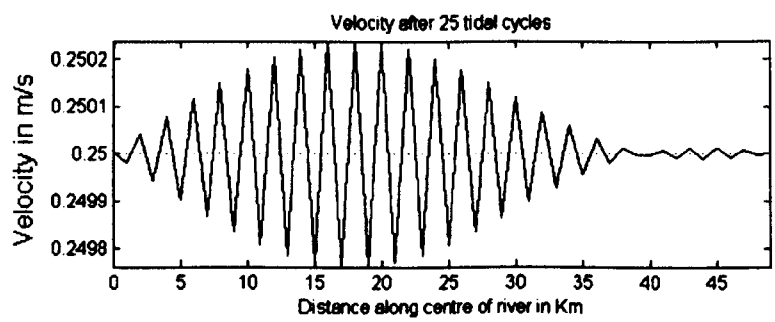

Figure 3.4b: Velocity under zero tidal flow

Figs. 3.4a and 3.4b depict the water depths and velocities at different locations in the River Thames after 25 tidal cycles. Although the graphs appear rather 'jagged', this is purely due

to the horizontal/vertical scaling. Classical Laplacian theory of hydrodynamics indicates that the water depth should level off to a steady state value of about 15.309 m (a personal communication from Johns [50]) which as can be seen above, the finite difference model does indeed do, oscillating about this mean value. Similarly, the velocity should also approach a mean of 0.25 m/s, again as depicted above. The figures represent a situation where there is no tidal inflow from the estuary whilst there is a constant inflow of 0.25 m/s at the river end, in other words, a steady state problem. A somewhat artificial case since there is always input into or output from the system at the estuarine end in the Thames. Although the scheme does not converge precisely to 15.309, it is stable with an oscillation about the mean of less than +/- 1 cm.

3.8 Data generation for flood tide flow

The following 'italicized' data is a typical example of the computer output and refers in particular to figs. 3.5a and 3.5b with the user response in bold:

Wave celerity = sqrt(gravity x still water depth).
So for still water depth = 20m, celerity = 14 m/s, for depth = 15m, celerity = 12.13 m/s, depth = 10m, celerity = 9.9m/s
As the number of required tidal cycles needs to be iterated through about ten times, then as a guide:
no. of cycles for 50km = 10, no. of cycles for 100km = 20, no. of cycles for 200 km = 40, no. of cycles for 600 km = 120 etc.
How many tidal cycles do you wish to iterate over (note 12.4 hours = one tidal cycle) ? ***25***
tidalcycles = 25
Enter no. of sections required (must be odd). e.g. 101 spans 50 Km, 201 spans 100 Km etc.) ? ***101***
Enter a value for the still water depth ***15***
*Enter a value for the upstream velocity .**25***
*Enter a value for the downstream velocity .**25***
Enter a value for the friction coefficient, say zero for no bottom stress or for example 0.0026
Value ? ***0***

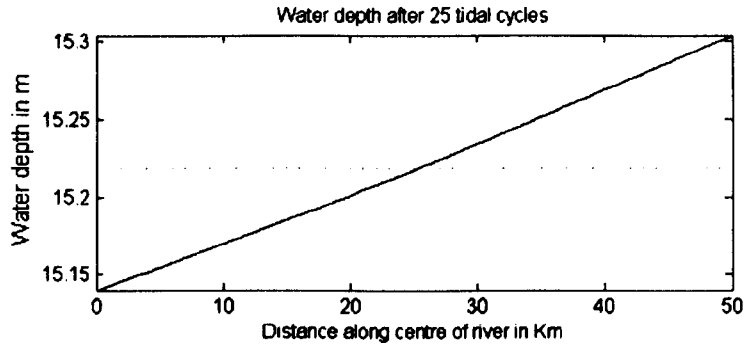

Figure 3.5a: Depth with flood tide flow of -0.25 m/s.

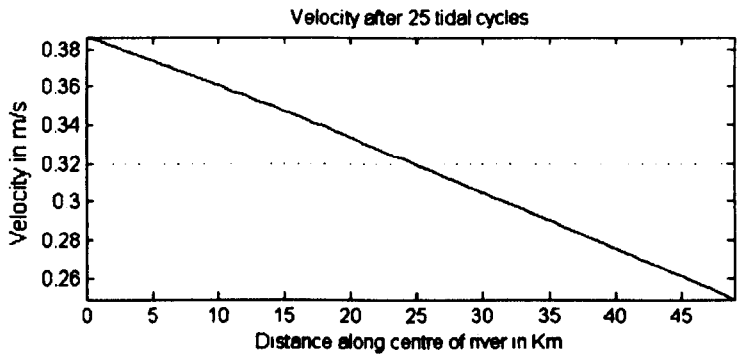

Figure 3.5b: Velocity with flood tide flow of -0.25 m/s.

Figs. 3.5a and b represent a configuration where there is an *opposing* forcing tidal inflow of -0.25 m/s at the estuary and a river input to the system at the x = 0 boundary of 0.25 m/s..

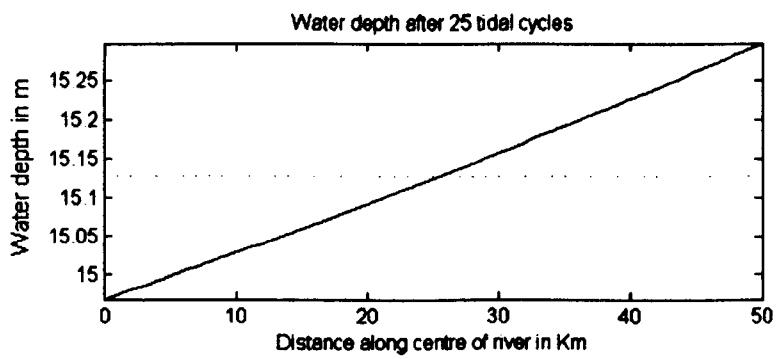

Figure 3.6a: Depth with flood tide flow of -0.5 m/s.

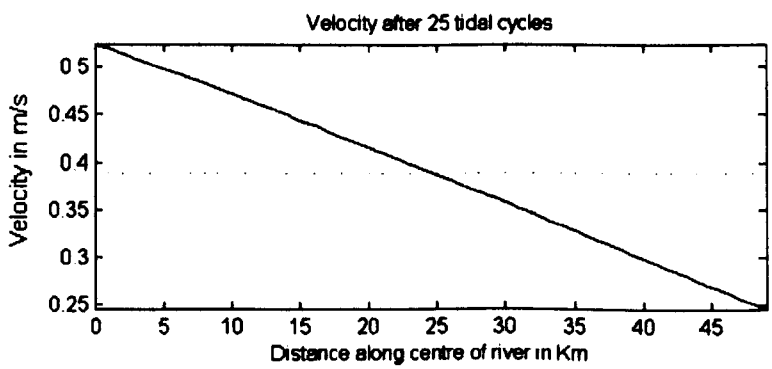

Figure 3.6b: Velocity with flood tide flow of -0.5 m/s.

Figs. 3.6a and b depict the case of an inflow of 0.25 m/s at the river end and a *larger* forcing inflow of -0.5 m/s at the estuarine end.

3.9 Data generation for ebb tide flow

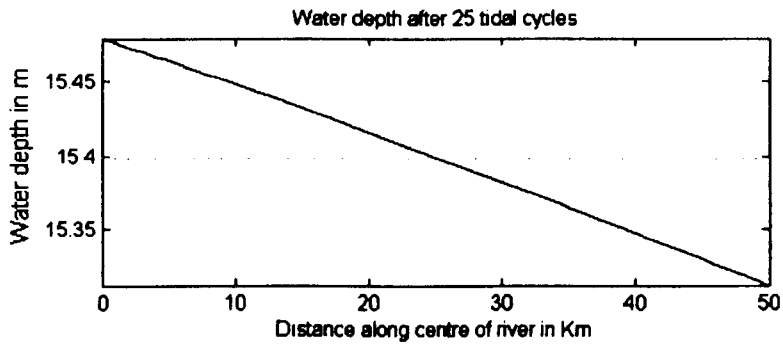

Figure 3.7a Depth with ebb tide flow of 0.25 m/s.

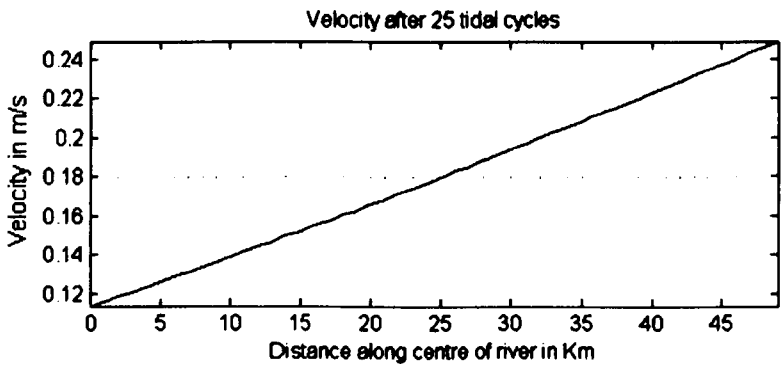

Figure 3.7b Velocity with ebb tide flow of 0.25 m/s.

Similarly, figs. 3.7a and b above represent the case where there is again a river inflow of 0.25 m/s but now a forcing tidal outflow of 0.25 m/s.

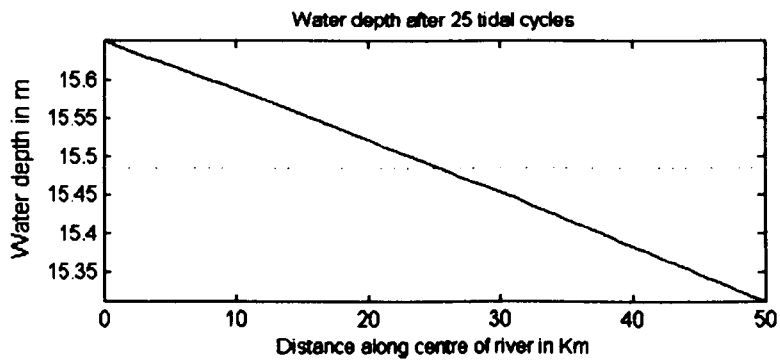

Figure 3.8a Depth with ebb tide flow of 0.5 m/s.

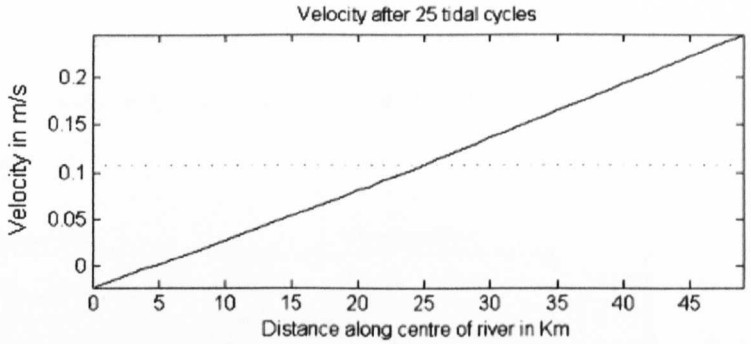

Figure 3.8b Velocity with ebb tide flow of 0.5 m/s.

In the last set of figures above, figs.3.8a and b, the scenario is similar to that of figs. 3.7, but now the forcing tidal velocity at the estuarine end is twice as large i.e. 0.5 m/s outflow.

3.10 Convergence of the Finite Difference Scheme

The next set of figures that follow demonstrate the *convergence* of the finite difference scheme at the upstream and downstream boundaries as well at the midpoint along the river as the number of tidal cycles of iteration is increased for various combinations of upstream and estuarine velocities. Figs. 3.9a and b depict the flow when the upstream boundary inflow is 0.25 m/s and the downstream forcing boundary flow (estuarine end) is -0.25 m/s.

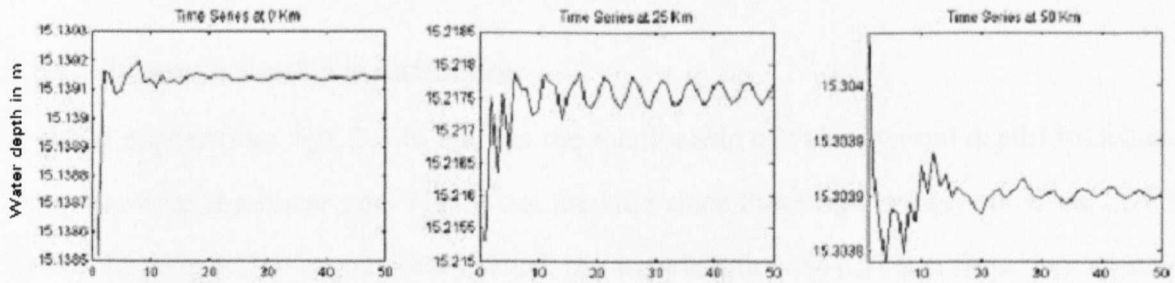

Figure 3.9a: Convergence of the FDS – flood tide

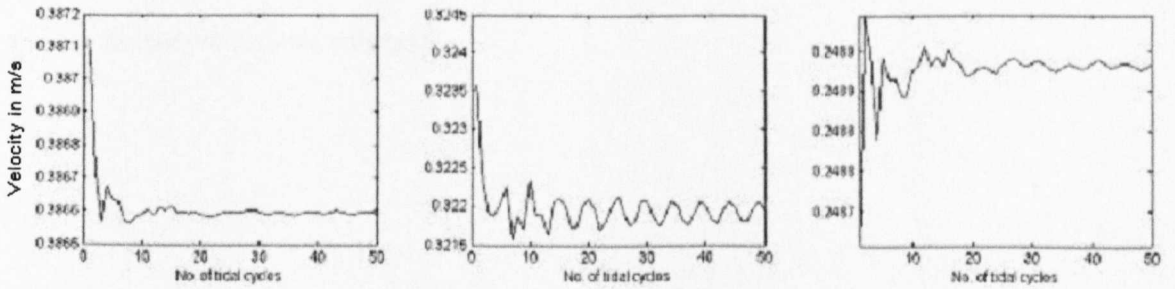

Figure 3.9b: Convergence of the FDS – flood tide

Correspondingly, figs. 3.10a and b represent the opposite flow, that is an ebb tide one where now the forcing velocity at the estuarine end although of the same magnitude, is now in the opposite direction.

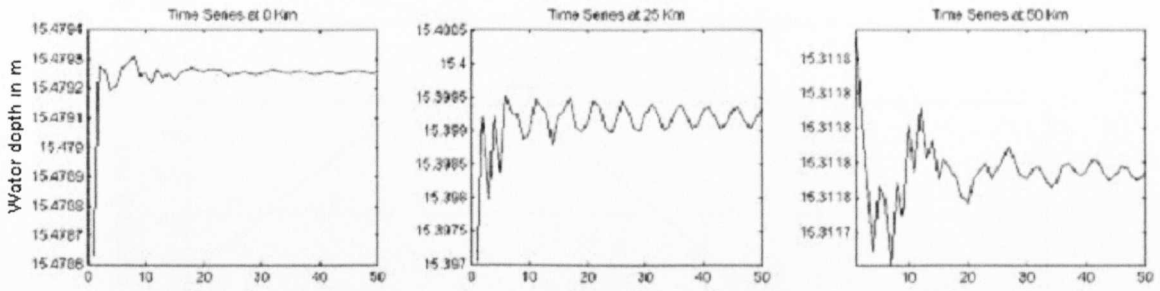

Figure 3.10a: Convergence of the FDS – ebb tide

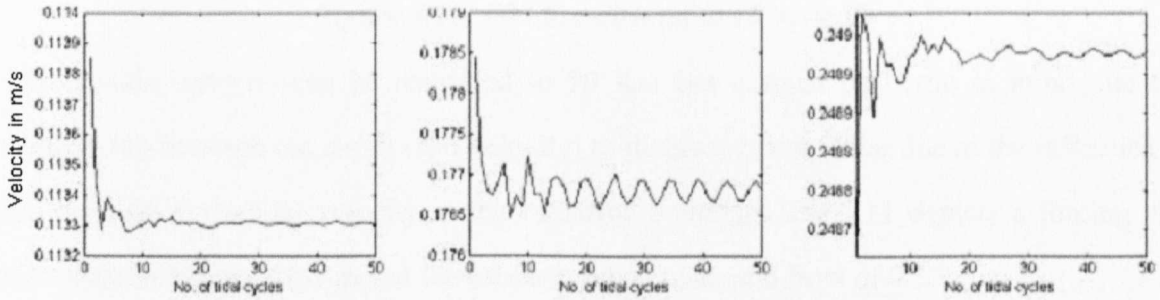

Figure 3.10b: Convergence of the FDS – ebb tide

3.11 Domain analysis extension

It would appear from figs. 3.5 to 3.8 that the relationship of velocity (and depth) to distance along the river is a linear one. This is not the case since these figures only show the first 50 Km of the river where as previously noted, the wavelength is 541.51 Km for a river depth of 15 m. Extending the domain analysis of the finite difference scheme to a length of 600 Km, a more extensive picture emerges:

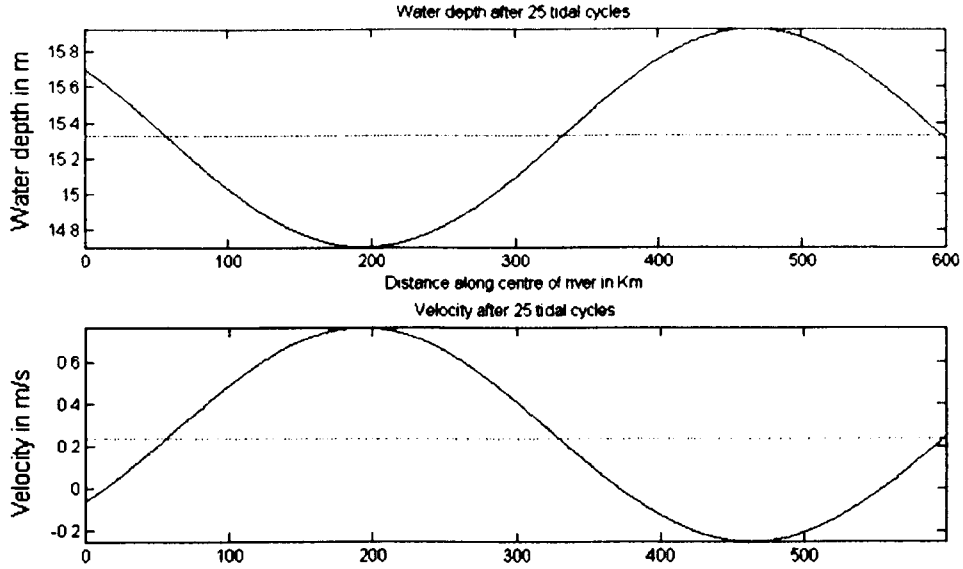

Figure 3.11: 600 Km domain at 15 m depth

The domain analysis can be restricted to 50 Km but it must be borne in mind that the relationship between the depth (and velocity) to distance is not linear due to the influence of the imposed sinusoidal velocity at the estuarine boundary. Fig.3.11 depicts a forcing ebb (downstream) flow of 0.5 m/s at the estuary and an upstream flow of 0.25 m/s.

The next set of figures depict the flows at a depth of 10 m and 20 m respectively with exactly the same values for the upstream and downstream forcing velocities:

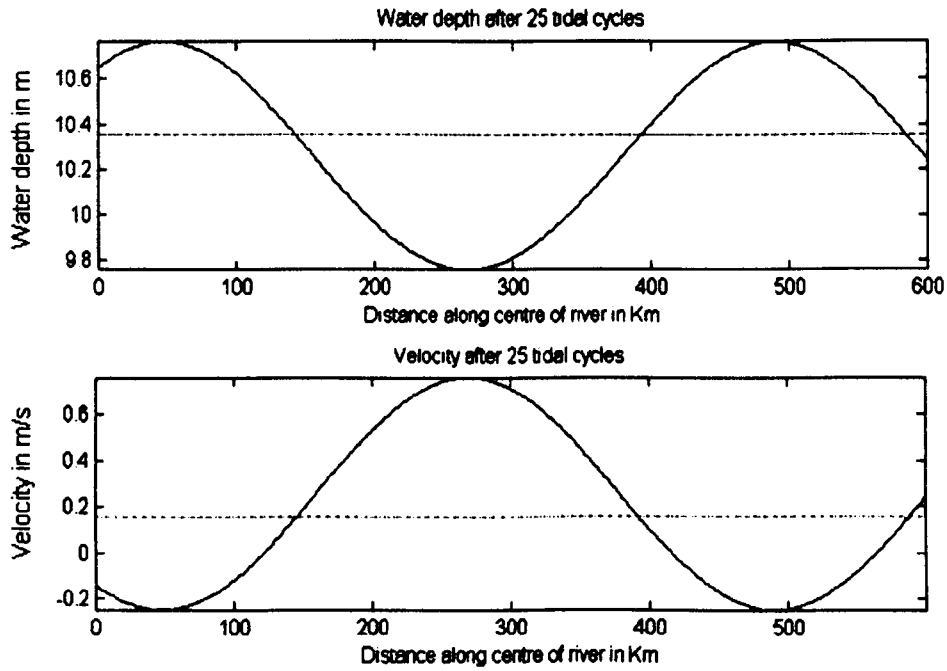

Figure 3.12: 600 Km domain at 10 m depth

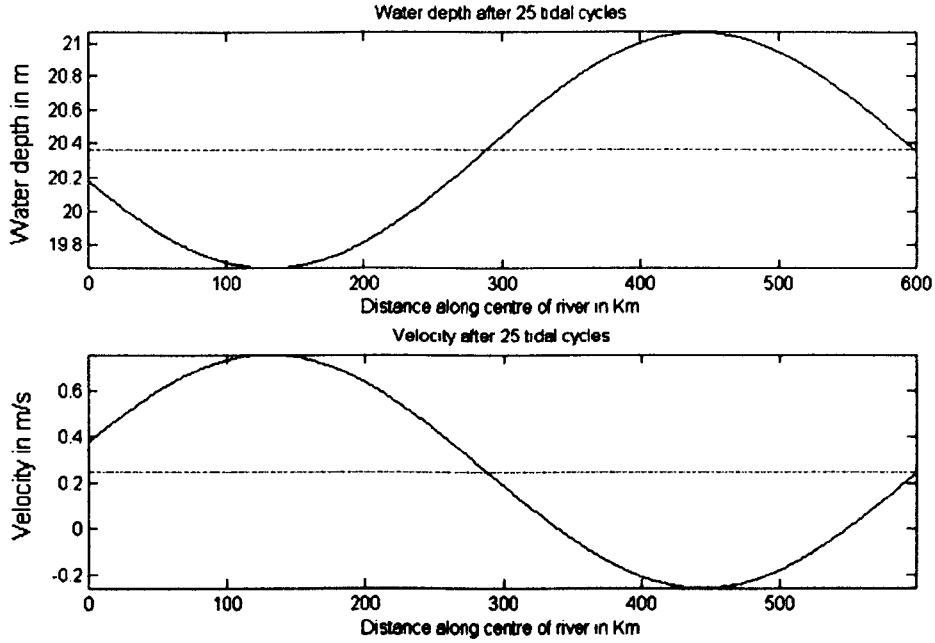

Figure 3.13: 600 Km domain at 20 m depth

3.12 Finite Difference Scheme – bottom friction model

So far the effects of bottom friction have been ignored. Bottom friction is essentially a resistance to movement of the flow. A main objective is to develop a neural network model incorporating these effects and so the research will return to an investigation of bottom friction in Chapter 5.

If the non-linear bed friction term is included, then equation **(3.6)** can, Vreugdenhil [106], be written:

$$\frac{\partial u}{\partial t} + g\frac{\partial \xi}{\partial x} + C_D \rho u|u| = 0 \qquad \textbf{(3.24)}$$

where C_D is a coefficient of friction and ρ is the density of seawater. Noting that the velocities 'u' in this bed friction term have to be depth averaged, this will give rise to a h^2 term. Provided the density is homogeneous throughout, that is a density of 1 (freshwater) is assumed, then the density term can be ignored. Hence, from equation **(3.24)**, equation **(3.10)** becomes

$$u_{i+1}^{j+1} = u_{i+1}^{j} - \frac{Cr}{2}\sqrt{\frac{g}{h}}\left(\xi_{i+2}^{j+1} - \xi_{i}^{j+1}\right) - C_D u_{i+1}^{j+1}\left|u_{i+1}^{j}\right|\frac{\delta t}{h^2}$$

which after a little manipulation becomes

$$u_{i+1}^{j+1} = \frac{u_{i+1}^{j} - \frac{Cr}{2}\sqrt{\frac{g}{h}}\left(\xi_{i+2}^{j+1} - \xi_{i}^{j+1}\right)}{1 + C_D\left|u_{i+1}^{j}\right|\frac{\delta t}{h^2}} \tag{3.25}$$

From a private communication with Johns [50], a suitable value for C_D is 0.0026. The FDS model to include bottom stress is therefore (repeating equation **(3.9)** for convenience):

$$\begin{aligned}\xi_i^{j+1} &= \xi_i^j - h\frac{\delta t}{2\delta x}\left(u_{i+1}^j - u_{i-1}^j\right)\\ &= \xi_i^j - \frac{c^2}{g}\frac{\delta t}{2\delta x}\left(u_{i+1}^j - u_{i-1}^j\right)\\ &= \xi_i^j - \frac{Cr}{2}\sqrt{\frac{h}{g}}\left(u_{i+1}^j - u_{i-1}^j\right)\end{aligned} \tag{3.9}$$

$$u_{i+1}^{j+1} = \frac{u_{i+1}^{j} - \frac{Cr}{2}\sqrt{\frac{g}{h}}\left(\xi_{i+2}^{j+1} - \xi_{i}^{j+1}\right)}{1 + 0.0026\left|u_{i+1}^{j}\right|\frac{\delta t}{h^2}} \tag{3.26}$$

subject to the boundary conditions **(3.22)** and **(3.23)**.

3.13 Data generation for the bottom friction model

To conclude this chapter, two further figures, figs. 3.14 and 3.15 depicting the effect of bottom stress on the flow follow. Although the effects of bottom friction in a neural network model will not be considered until Chapter 5, it is worth noting the effect of bottom friction at this stage and hence its importance in river flows. Fig. 3.14 depicts an ebb flow with an upstream velocity of 0.25 m/s and a downstream velocity of 0.5 m/s. Contrastingly, fig. 3.15 shows the flow with the same magnitude velocities but now in a flood tide mode. As can be seen from the figures, there is a tendency for the bottom friction to 'dampen' the properties of the flow. As a result of the 'resistance' due to this friction, the deviations of the wave height from the still water level are reduced.

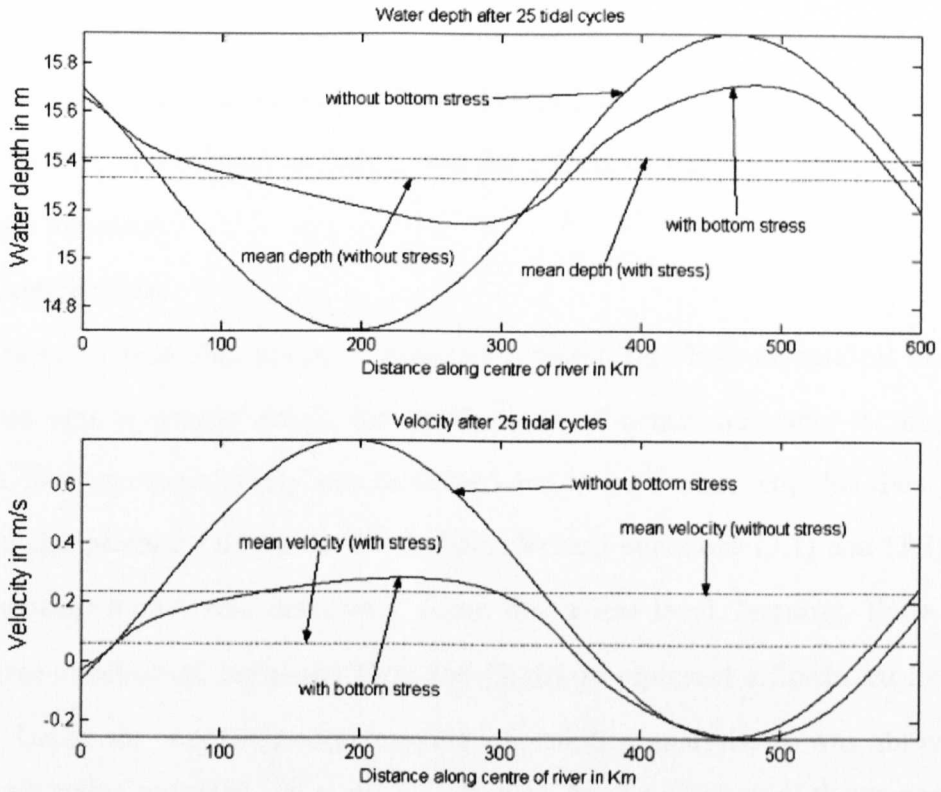

Figure 3.14: Effects of bottom stress during ebb flow

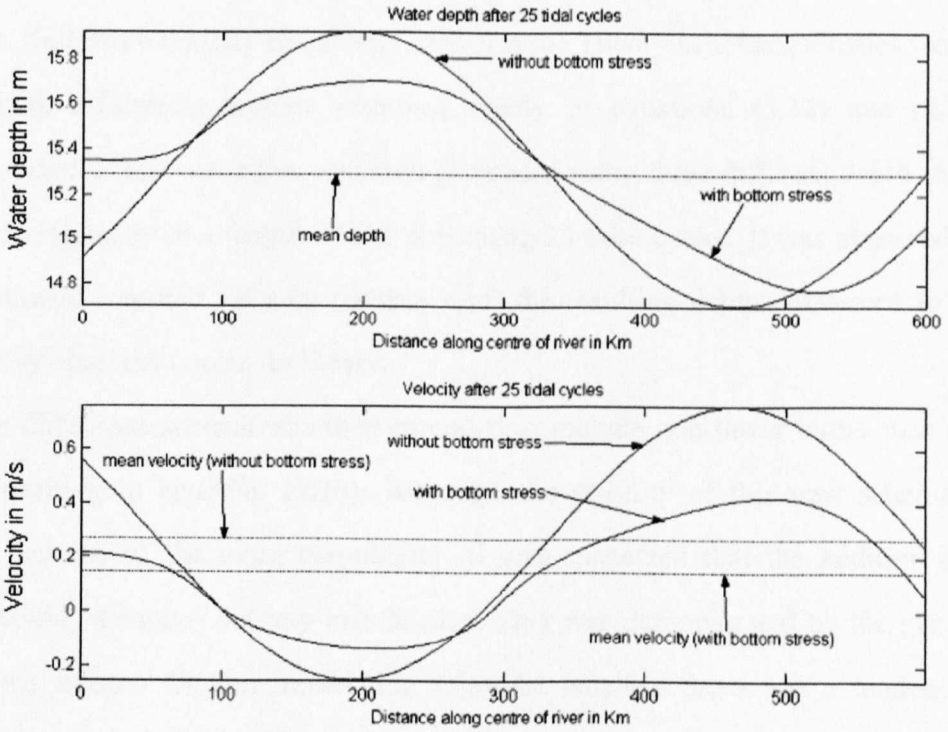

Figure 3.15: Effects of bottom stress during flood flow

The bottom friction effectively due to the drag effect, tends to work in the opposite direction of the flow. This results in the velocities being restricted to a narrower range of values. It

should be noted that this damping effect is proportional to the value of C_D so that if a larger value was chosen, the damping effect shown in figs 3.14 and 3.15 would be even greater. It can also be seen from equation **(3.16)**, that the effects of bottom friction diminish as the water depth increases.

3.14 Conclusion

In this chapter a brief discussion of previous research into hydrodynamical modelling of river flows and in greater detail, the applications of neural networks to this field was provided. Shallow water theory was described and in particular, why this theory could be applied in this research. Based on the de Saint Venant equations **(3.1)** and **(3.2)**, a simple hydrodynamical model was developed. From this a one level, leapfrog, finite difference scheme was constructed, equations **(3.9)** and **(3.10)**, to represent a linearized flow (no bed friction). Using the Von Neumann method of stability analysis, it was shown that the scheme was stable provided $|\rho|^2 \equiv \rho\bar{\rho} = 1$ and also, by examination of the truncation errors in the scheme, the latter was indeed found to be compatible with the original differential equations. Suitable boundary conditions based on the Theory of Characteristics were applied to the finite difference scheme resulting finally in equations **(3.22)** and **(3.23)**. Data containing depths and velocities was then generated using three different depth scenarios in an ebb tide regime over a length of 600 Km using 25 tidal cycles. It was observed that with no downstream imposed velocity (estuary end) the resulting depth converged to the value predicted by classical Laplacian theory.

The finite difference scheme was then extended to include non-linear terms such as bottom friction resulting in equation **(3.26)**. Although the stability of this new scheme was not analysed because of the extra complexity, it was suspected that the addition of bottom friction would 'dampen' out any instabilities. This was demonstrated by the generation of data for the bottom friction model that followed wherein there was a tendency for the friction to level out both the depths and velocities. Figs. 3.14 and 3.15 depict the significant effects of bottom friction in modelling river flows.

It should be emphasised that there are finite difference schemes other than that specified by equations **(3.9)** and **(3.26)** that involve implicit as well as explicit terms or schemes using

two initial time levels rather than one. However, one of the objectives of this research is to test the feasibility of using a neural network in a scenario involving bed friction (Chapter 5) and not a comparison of different finite difference methods. Notwithstanding this, the scheme compared favourably with the numerical model of Johns [51] (Chapter 4).

Chapter 4
Neural network model with zero bottom friction

4.1 Introduction

In this chapter the research concentrates on the development of a neural network to simulate the depths and velocities of a hydrodynamical model without bottom friction (cf. Chapter 3). Although the ultimate goal is the development of a model including bottom friction, the approach adopted here is to obtain some idea as to the architecture and other parameters involved in this simpler structure. In this chapter the methodology of the backpropagation training algorithm is discussed and the various measures of error are described. Various training and validation simulations are performed to assess the most suitable network architecture, and this is further refined by attempting to determine the optimum values for other parameters such as the momentum term and the learning rate. Finally, a comparison of the neural network simulations is made with a numerical model of Johns [51].

4.2 Model neural network architecture

The type of ANN to be used to model the hydrodynamical problem, discussed in Chapter 3, is a multilayer perceptron (MLP). This form of ANN architecture was briefly analyzed in Chapter 2 and in particular, depicted in fig. 2.7. It was decided to use a MLP network since it is frequently used by researchers in the hydrodynamics field and most importantly, used in one of the main papers referenced in this research and so readily available for comparison. As much of the work using measured hydrodynamic data involves some form of regression analysis whilst MLPs themselves are closely related to statistical regression, Sarle [92], this form of architecture was a natural choice. Further, MLPs are very good at learning functions that contain little or no discontinuities. Generally, hydrodynamic equations are if anything, 'smooth' and continuous mathematically.

As can be seen from the hydrodynamical model finite difference schemes equations **(3.9)** and **(3.10)**, from the previous chapter, repeated here for clarity, there will be three inputs and just one output:

$$\begin{aligned}\xi_i^{j+1} &= \xi_i^j - h\frac{\delta t}{2\delta x}\left(u_{i+1}^j - u_{i-1}^j\right)\\ &= \xi_i^j - \frac{c^2}{g}\frac{\delta t}{2\delta x}\left(u_{i+1}^j - u_{i-1}^j\right)\\ &= \xi_i^j - \frac{Cr}{2}\sqrt{\frac{h}{g}}\left(u_{i+1}^j - u_{i-1}^j\right)\end{aligned} \tag{4.1}$$

$$\begin{aligned}u_{i+1}^{j+1} &= u_{i+1}^j - g\frac{\delta t}{2\delta x}\left(\xi_{i+2}^{j+1} - \xi_i^{j+1}\right)\\ &= u_{i+1}^j - \frac{c^2}{h}\frac{\delta t}{2\delta x}\left(\xi_{i+2}^{j+1} - \xi_i^{j+1}\right)\\ &= u_{i+1}^j - \frac{Cr}{2}\sqrt{\frac{g}{h}}\left(\xi_{i+2}^{j+1} - \xi_i^{j+1}\right)\end{aligned} \tag{4.2}$$

Hence an ANN structure of three inputs and one output was adopted for the modelling. By Kolmogorov's theorem (cf. Chapter 2), Marques de Sá [74], Hornik [45], provided there are enough neurons in the hidden layer, only one hidden layer (or at most two) should be sufficient to ensure that the structure has the properties of a 'universal approximator' for this particular problem. Further, Marques de Sá [74], the number of neurons in the hidden layer would most likely be 2N + 1 where N is the number of neurons in the input layer. If there are too few hidden units there will be both a high training error and generalization error due to underfitting and high statistical bias. Correspondingly, too many hidden units result in a low training error but still in a high generalization error due to overfitting and high variance. Hornik [45] indicates that for the MLP to have this universal approximation property where there are hidden layers in the architecture, then there must be bias terms present. However, he does provide the caveat that in the case of the usual sigmoid activation functions, a fixed non zero bias term can be used instead of a trainable bias term.

There will in fact be two networks to develop as can be seen from equations **(4.1)** and **(4.2)** above. The first network will have the inputs ξ_i^j, u_{i+1}^j and u_{i-1}^j with an output of ξ_i^{j+1}, the updated depth that will feed into the second network. The inputs to this second network will then be u_{i+1}^j, ξ_{i+2}^{j+1} and ξ_i^{j+1} from which the output u_{i+1}^{j+1}, the updated velocity, is obtained. A possible network architecture is therefore outlined below in fig. 4.1, the actual number of neurons in the hidden layer to be verified during the course of this research.

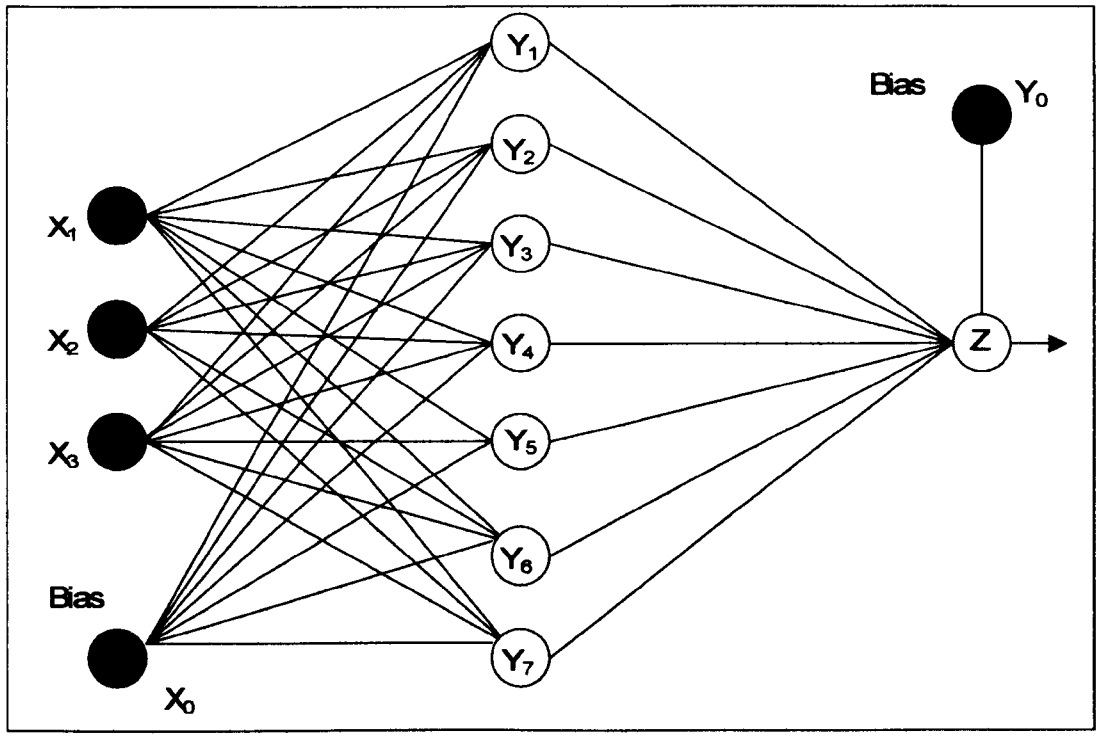

Figure 4.1: Possible network architecture

In relation to the fig. 4.1 above, for the first (depth) network, the inputs are

$$\xi_i^j \to x_1, \quad u_{i-1}^j \to x_2, \quad u_{i+1}^j \to x_3, \quad z \to \xi_i^{j+1}$$

and for the second (velocity),

$$u_{i+1}^j \to x_1, \quad \xi_i^{j+1} \to x_2, \quad \xi_{i+2}^{j+1} \to x_3, \quad z \to u_{i+1}^{j+1}$$

That is, the first network is trained using a combination of two previous velocities and one previous depth with the target value being the latest depth. Conversely, in the second, the training uses a combination of one previous velocity and two latest depths where the target value is now supplied by the latest velocity.

4.3 Training Schema

The complete scheme is outlined in fig. 4.2:

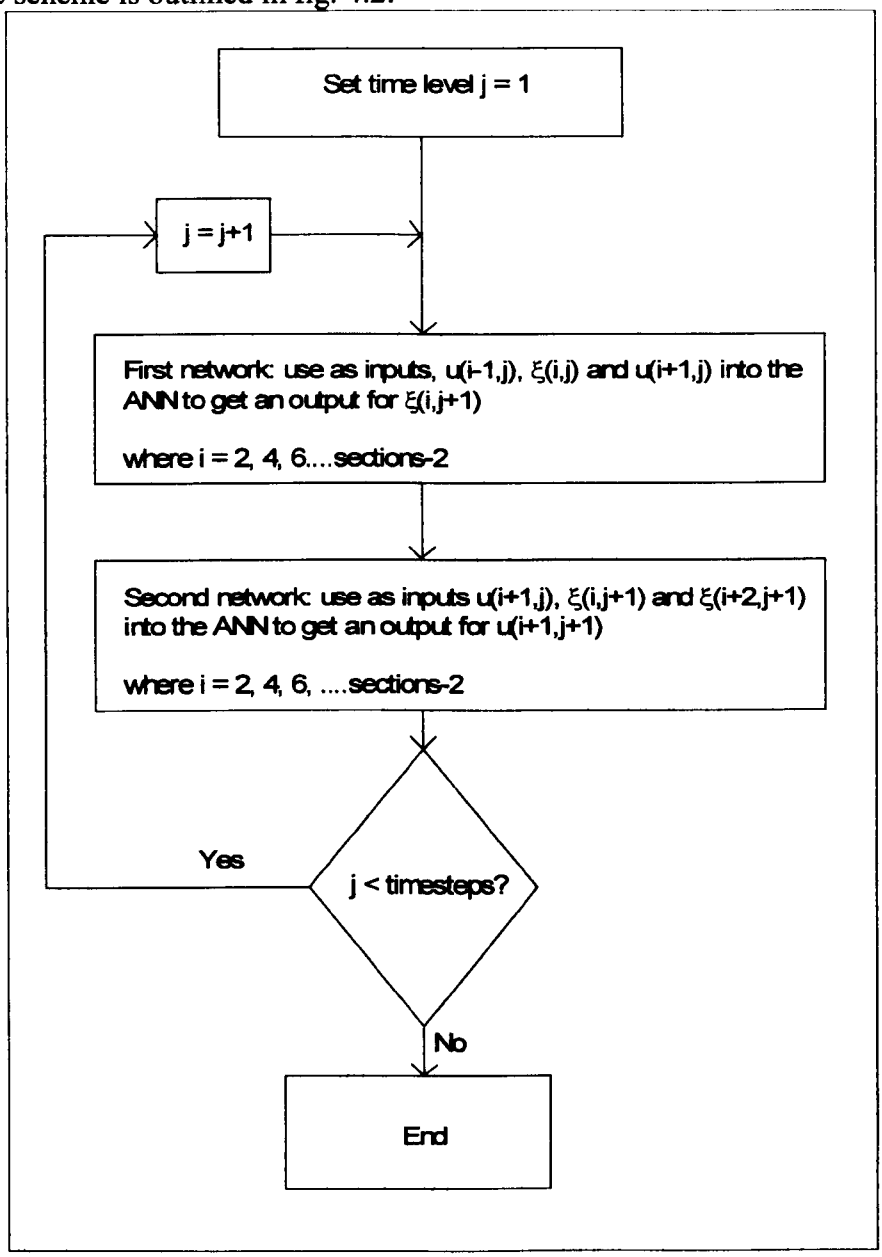

Figure 4.2: Training schema

With respect to the training of the network, this was performed using the backpropogation algorithm in incremental mode rather than batch mode for the possibly faster (in terms of convergence) second derivative methods such as standard conjugate gradient or Levenberg-Marquardt algorithms could not be used since they are essentially batch algorithms. The algorithm was applied with learning rates and momentum terms and initially, the bi-polar activation function.

Although the model equations **(3.3)** and **(3.4)** have been linearized to first order so that in principle a simple linear threshold would suffice, the addition of the non-linear boundary conditions **(3.20)** and **(3.21)** as well as the non-linear bed friction terms require a non-linear threshold function. The bi-polar function, which was chosen (as mentioned) as its output is restricted to the range [-1, 1] which matches that precisely of the depths and velocities since the depth variation $\xi \in$ [still water level $\pm$ 1] and u $\in$ [mean velocity $\pm$ 1]. In fact, as the amplitude of the wave $A = U\sqrt{\frac{h}{g}}$, it can easily be seen that for still water depths of 20 m or less and a velocity U of 1 m/s, the maximum amplitude would be +/- 0.7 m from the still water level. Exceptions to this of course would be situations involving 'bores' such as the famous one that travels up the Severn Estuary. The outputs from the finite difference model therefore fall within the range [-1, +1]. In practical terms therefore, it was better to use this activation function [bi-polar] rather than the sigmoid and since the inputs were of the same order of magnitude as the target, no scaling of the inputs or outputs was therefore necessary. In fact, it is not unusual to not scale the data, for example, in the neural network modelling of time series. The equations to perform such scaling, equations **(2.9)** and **(2.10)** were briefly discussed in Chapter 2. However, this was investigated using these scaling equations and it was observed that much better agreement between the target and output variables was obtained when scaling was not employed. Further, it was observed on analysis that all of the variable values fell within the bipolar range when, with U_0 = river initial velocity and U_L = initial estuarine velocity (cf. Chapter 3):

still water depth = 20 m, no bed stress

$$|U_0| + |U_L| \le 0.7$$

with bed stress

$$|U_0| + |U_L| \le 0.7, \qquad |U_0| \le 0.4$$

still water depth = 15 m, no bed stress

$$|U_0| + |U_L| \le 0.8$$

with bed stress

$$|U_0| + |U_L| \le 0.8, \qquad |U_0| \le 0.5$$

still water depth = 10 m, no bed stress

$$|U_0| + |U_L| \leq 0.95$$

with bed stress

$$|U_0| + |U_L| \leq 0.95, \qquad |U_0| \leq 0.4$$

4.4 Backpropagation Algorithm

Most of the concepts used in the rest of this chapter i.e. activation functions, hidden layers, learning rates, momentum terms, underfitting, overfitting, incremental and epoch based learning were discussed in Chapter 2 and so will not be repeated here.

The backpropagation technique is essentially a gradient descent process and is one of the most widely used algorithms for training a MLP. In the MLP, a feedforward network as discussed in Chapter 2, the connections between the different layers are bi-directional in the sense that errors can be propagated back from the output layer to the input layer so that all weights are modifiable. Essentially a pattern of weighted inputs is presented to a hidden layer (if there is one), activated upon via the transfer function and then forwarded on with new weights to the output layer where these 'inputs' are further activated upon using the same (but not necessarily so) transfer function. The differences between the desired sample (or target) outputs and the actual outputs are then used to generate 'error signals' which can be used to modify the weights between the output and hidden layers, and then subsequently between the hidden and input layers. The ultimate goal of the process is to reduce these error signals to an acceptably low value.

The rule by which this is accomplished is called the 'Widrow-Hoff' rule or 'Least Squares Rule' (LMS) or 'Generalized Delta Rule'. The latter name refers to the fact that the original 'delta rule' was applicable only to a single layer perceptron (as mentioned in Chapter 2) and not to a MLP and so had to be amended. The full mathematical derivation of this rule can be found in any standard text on neural networks such as Haykin [43], Picton [85], Tarassenko [99] et. al. The process of pattern presentation to the network is repeated many times, each time errors signals being generated and the weights adjusted accordingly. When the error signals approach an acceptably small value, the so called training is stopped and the weights are then 'frozen'. To aid convergence to these acceptable values and to avoid local minima

and overfitting, learning rates and momentum terms are added to the mathematical formulation of the rule.

Consider a MLP consisting of just one hidden layer such that there are in total, N input units, L hidden units and W output units represented by the subscripts i, j and k respectively for each set of training patterns presented to the network. It should be noted that different authors use a different ordering of the subscripts i, j, k on the weights particularly on the backward pass through the algorithm. Gurney [38], Browne [14], Haykin [43], Skapura [96] for example use the ordering kj and ji whilst Tarassenko [99] and Callan [17] use jk and ij. Whichever annotation is used, it is important to be consistent in terms of the weight matrices and their transpose. The following algorithm (with some important modifications) is based on the work of Browne [14].

4.5 Application of the backpropagation algorithm

4.5.1 Forward Pass

- present a net (weighted) input to each unit j in the hidden layer

$$net_j = \sum_{i=1}^{N} w_{ij} x_i \tag{4.3}$$

where $\{x_i\}$ represent {input units}, $\{w_{ij}\}$ represents {weights from input unit i to hidden unit j}.

- Pass this net result through a transfer (activation) function

$$Oh_j = \int(net_j) \tag{4.4}$$

for each of the hidden units j, where the " $\int$ " means activation (not integration in the standard sense of calculus) to give an activated result Oh_j.

- Present the set of weighted (using new weights) activated outputs from the hidden layer to the output layer.

$$net_k = \sum_{j=1}^{L} w_{jk} Oh_j \tag{4.5}$$

- This results in the following:

$$Oo_k = \int(net_k) \tag{4.6}$$

which are the activated values of the 'inputs' to the output layer.

4.5.2 Backward Pass

The activated outputs $\{Oo_k\}$ from step (4) are compared with the desired target activation values $\{d_k\}$ from which the error signals are generated and further, from which a δ quantity (hence the 'delta' rule) can then be calculated for each of the output units.

- For each output unit,

$$\text{error signal} = (d_k - Oo_k) \tag{4.7}$$

- The δ term for each output unit is

$$\delta o_k = (d_k - Oo_k)Oo_k(1 - Oo_k) \tag{4.8}$$

- As for the output layer, error signals are then calculated for each unit in the hidden layer.

$$\text{error signal} = \sum_{k=1}^{W} \delta o_k w_{jk} \tag{4.9}$$

- from which can be calculated the δ term for each of the hidden units

$$\delta h_j = (Oh_j)(1 - Oh_j)\sum_{k=1}^{W} \delta o_k w_{jk} \tag{4.10}$$

- To change the weights between the output and hidden layers, weight error derivatives are calculated for each of the weights between the output and hidden layers according to the formula

$$wed_{jk} = \delta o_k Oh_j \tag{4.11}$$

The weights are then adjusted using the formula

$$w_{jk}(t+1) = w_{jk}(t) + \eta wed_{jk} \tag{4.12}$$

where $w_{jk}(t+1)$ represents the adjusted weight to be used in the next iteration of the forward pass. The quantity η is the learning rate. If a momentum term α is included, the equation is:

$$w_{jk}(t+1) = w_{jk}(t) + \eta wed_{jk}(t) + \alpha wed_{jk}(t-1) \tag{4.13}$$

so that now the weight error derivative from the previous cycle is used as well.

If some form of regularization is employed (to aid convergence) such as weight decay, Bishop [9], then equation **(4.13)** becomes:

$$w_{jk}(t+1) = w_{jk}(t) + \eta wed_{jk}(t) + \alpha wed_{jk}(t-1) - d_3 wed_{jk}(t) \tag{4.14}$$

where d_3 is the (one) decay constant from the units in the hidden layer to the output layer.

If a bias unit is included, remembering that the activation of a bias unit is usually 1, the weight error derivative for the bias is

$$wed_k = \delta o_k \tag{4.15}$$

with the weights being adjusted by:

$$w_k(t+1) = w_k(t) + \eta wed_k(t) + \alpha wed_k(t-1) \tag{4.16}$$

If again, weight decay is employed, (4.16) becomes:

$$w_k(t+1) = w_k(t) + \eta wed_k(t) + \alpha wed_k(t-1) - d_4 wed_k(t) \tag{4.17}$$

where d_4 is the (one) decay constant from the bias to the output layer.

- Similarly, weight error derivatives are calculated for the weights between the hidden and input layers:

$$wed_{ij} = \delta h_j x_i \tag{4.18}$$

and these weights between the hidden and input layers are amended in a similar fashion using

$$w_{ij}(t+1) = w_{ij}(t) + \eta wed_{ij} \tag{4.19}$$

or

$$w_{ij}(t+1) = w_{ij}(t) + \eta wed_{ij}(t) + \alpha wed_{ij}(t-1) \tag{4.20}$$

or

$$w_{ij}(t+1) = w_{ij}(t) + \eta wed_{ij}(t) + \alpha wed_{ij}(t-1) - d_1 wed_{ij}(t) \tag{4.21}$$

For bias units attached to the hidden layer, in a similar fashion as for the output layer,

the weight error derivative is:

$$wed_j = \delta h_j \tag{4.22}$$

and the weight adjusted by:

$$w_j(t+1) = w_j(t) + \eta wed_j(t) + \alpha wed_j(t-1) - d_2 wed_j(t) \tag{4.23}$$

It should be noted that the four decay constants do not have to be necessarily the same and also, equations **(4.8)** and **(4.10)**, as written, are only valid when the transfer function is the sigmoidal (logistic) one since if

$$f(x)=\frac{1}{1+e^{-x}} \quad \text{then} \quad f'(x)=f(x)[1-f(x)].$$

If however, the transfer function is the tanh function, then with

$$f(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \quad \text{we have} \quad f'(x)=1-[f(x)]^{2}.$$

This implies that equations **(4.8)** and **(4.10)** must, if using the tanh transfer function, be amended to:

$$\delta o_k=(d_k-Oo_k)\left(1-[Oo_k]^2\right) \tag{4.24}$$

$$\delta h_j=\left(1-[Oh_j]^2\right)\sum_{k=1}^{W}\delta o_k w_{jk} \tag{4.25}$$

respectively.

The training of a neural network using the backpropagation algorithm was briefly discussed in Chapter 2, where concepts such as local minima, incremental updating and training sets were introduced. It should be noted that if a weight decay is employed then the inputs and targets usually have to be standardized and the bias terms preferably omitted, Sarle [92].

4.6 Error Measures

There are essentially four measures of error considered in this work: the *mean error* which gives some measure of the bias on each output and the *root mean square error* (RMSE) which is basically an indicator of the strength of relationship between the input and output as well as the amount of *noise* in the data, the *error standard deviation* and also the *relative percentage error*. With $e_i=z_i-t_i$ = the individual errors of n patterns where z_i, t_i represent the outputs and targets respectively, then the four following error measures, equations **(4.26)** to **(4.29)** are:

$$\text{mean (absolute) error } M_e=\frac{1}{n}\sum_{i=1}^{n}|e_i| \tag{4.26}$$

Sometimes referred to as the mean absolute deviation, it is usually similar in magnitude to, but slightly smaller than, the root mean squared error. If the data sets are small or limited, the mean (absolute) error is the preferred method. In contrast, the mean error (that does not

use absolute values) is not a measure of accuracy since large errors of equal magnitude but opposite sign can cancel each other out.

$$\text{root mean squared error} = \pm\sqrt{\frac{1}{n}\sum_{i=1}^{n} e_i^{\,2}}\,. \tag{4.27}$$

This is a measure of dispersion and in regression analysis is referred to as the standard error of the estimate. It is more sensitive than other measures to the occasional large error since the squaring process gives disproportionate weight to very large errors. Squaring equation **(4.27)** and multiplying by n gives the 'total squared error loss' frequently abbreviated in statistics to 'SSE'. If equation **(4.27)** was just simply squared, the 'mean squared error loss' often abbreviated to 'MSE' is obtained. In classification problems in the field of neural networks, the MSE is often referred to as 'Empirical Risk'.

$$\text{error standard deviation } s_e = \pm\frac{1}{(n-1)}\sqrt{\sum_{i=1}^{n}(e_i - M_e)^2} \tag{4.28}$$

$$\text{relative percentage error: } E_{rel} = \frac{1}{n}\sum_{i=1}^{n}\left\{\frac{|e_i|}{|t_i|}\right\}\times 100\%\,. \tag{4.29}$$

The latter is also sometimes called the mean absolute percent error and if the error is large, it implies the 'fit' is not a particularly good one. By subtracting this value from 100, a type of measure of 'correctness' in percentage terms is derived. With σ^2 representing the variance of the errors, the mean error and the root mean squared error are related in such a way that

$$\frac{1}{n}\sum_{i}^{n} e_i^2 = \sigma^2 + \left(\frac{1}{n}\sum_{i}^{n} e_i\right)^2 \tag{4.30}$$

For the sake of brevity, within tables in both this chapter and the next (Chapter 5), these error measures will be abbreviated to: ME, RMSE, ESD and RPE. It should be noted that many researchers just consider the one error measure, RMSE. To complicate matters further, sometimes the square root is not taken so that the values being quoted are actually the MSE (mean squared error).

In the work that follows, it was observed that generally:

$$\frac{ME}{RMSE} \sim \pm 0.75, \quad \frac{ESD}{RMSE} \sim \pm 0.03 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5$$

It is noted that for the last ratio, the units have not cancelled out and so for the depth analysis it will be m^{-1} and for the velocity analysis, $m^{-1}s$.

4.7 Software programs used in the simulation

There are five Matlab programs used in this research:

VenantDataGenerator2 – generates the depths and velocities (with or without bottom stress) from the finite difference schemes as discussed in Chapter 3.

DepthSolution – trains the neural network using back propagation on the depths and velocities generated by the previous program, VenantDataGenerator2, to obtain a simulation of the depths. This program is also used to select between different architectures i.e. the no. of hidden units

DepthSolutionValidation – (using fixed weights pertaining to each of the different architectures) validates the different architectures against an unseen data set (re-created by VenantDataGenerator2.) This program is also used once more using a third re-created (and unseen) data set to test the final selected network over a reduced data set i.e. 60 Km (see Chapter 5 Section 5.6, in particular, sub section 5.6.1).

VelocitySolution – trains the neural network on the velocities and depths generated by DepthSolution to obtain a simulation of the velocities. This program is also used to select between different architectures.

VelocitysolutionValidation – as for DepthSolutionValidation.

4.8 Training the depth network (zero bottom friction)

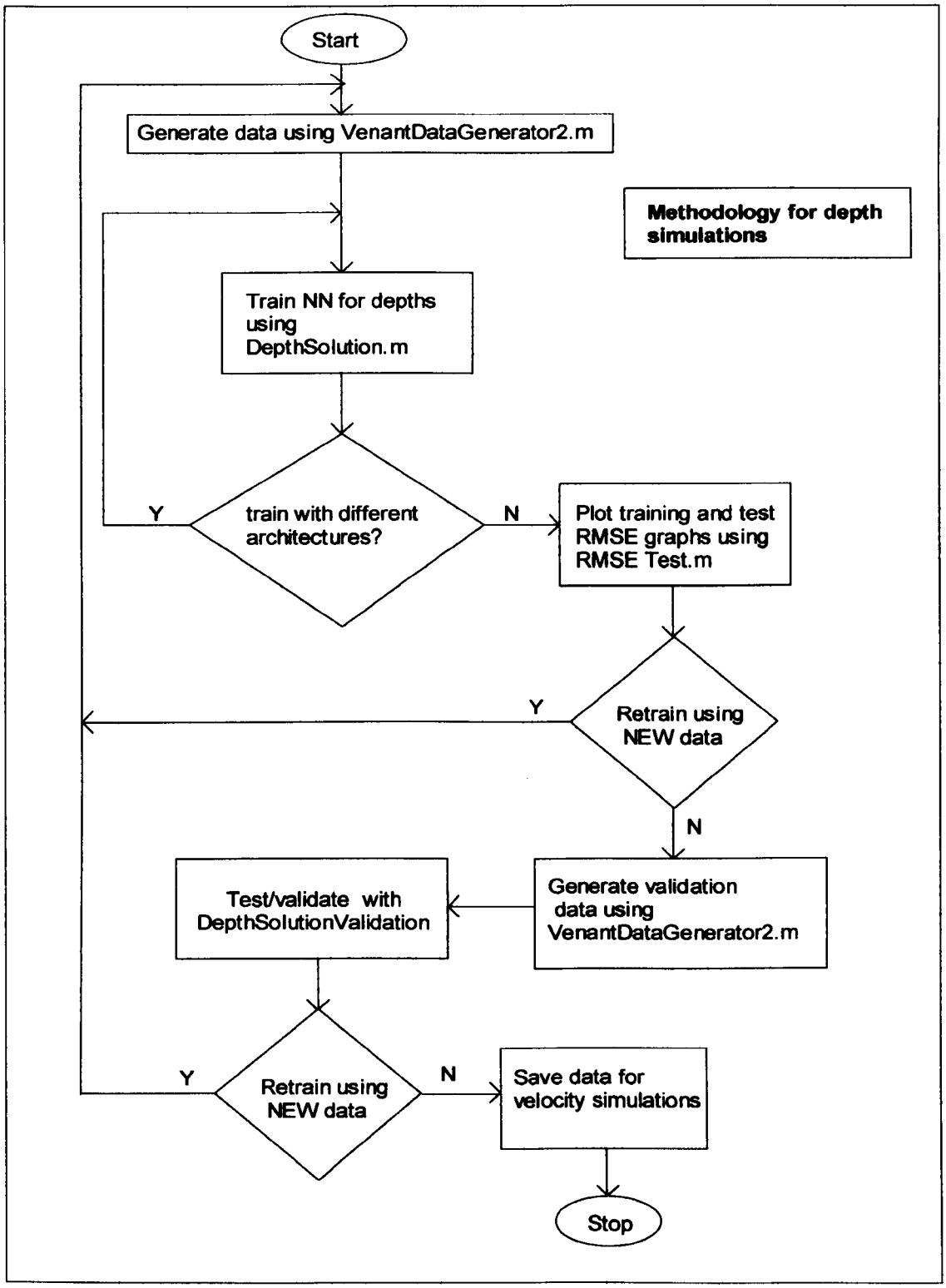

Figure 4.3: Training and testing methodology

The overall methodology of the depth training and testing is depicted in fig. 4.3.

The first network, that is the one for solving for the depths as indicated by equation **(4.1)** and fig. 4.1, was trained using data generated by the finite difference schemes outlined in Chapter 3. The Matlab program to generate this data is *VenantDataGenerator2* as mentioned in the previous section (cf. Section 4.7) whilst the Matlab program to perform the training is *DepthSolution*. Both programs are listed in appendix C. The latter program also has the facility to either have the weights entered manually or entered automatically by MATLAB using the random number generator. The automatic process was the preferred method.

4.8.1 Pseudo Code of the Depth training program

Brief pseudo code of the *DepthSolution* program follows:

Filename: DepthSolution

***COMMENT**: USE THIS PROGRAM ALSO FOR VALIDATING DIFFERENT NEURAL NETWORK ARCHITECTURES. This is a single hidden layer MLP using the sigmoid or tanh activation functions and the backpropagation algorithm. THIS PROGRAM SOLVES FOR THE HEIGHTS ABOVE THE STEADY STATE WATER LEVEL. THIS PROGRAM USES THE DATA FILE VenantOriginal to acquire the original data created by VenantGenerator2*

***PROCESS**_get inital data*

***GET** type of activation function, learning rate and momentum term*

***GET** desired rmse, no. of neurons in hidden layer and no. of iterations*

***LOAD** file containing hydraulic data computed using the VenantDataGenerator2*

END

***PROCESS**_initalize arrays*

***INITIALIZE** pattern, input layer and output layer arrays*

***INITIALIZE** arrays to hold weights, weight error derivatives, delta adjustments, activation results, targets and error signals*

***INITIALIZE** variables bias, iteration counter, mean error, max rmse and outer counter*

END

```
PROCESS_get weights
    SWITCH_enter weights manually?
        CASE weights entered automatically by system
            create matrices of randomized weights
        OTHERWISE
            LOOP
                enter weights for input layer to hidden layer
            END
            LOOP
                enter weights for hidden layer to output layer
            END
            LOOP
                enter bias weights to hidden layer
            END
            LOOP
                enter bias weights to output layer
            END
        END
    END
END
PROCESS_copy
    COPY weight matrices
END
PROCESS_transfer data
    LOOP
        fill target matrix witth data from file
        fill pattern matrix with data from file
    END
END
PROCESS_start of main computation
    LOOP_ outer loop using outercounter
        LOOP_ inner loop using innercounter
            select input pattern
```

select target pattern

***PROCESS**_start of forward pass*

pass weighted input data to the hidden layer

apply chosen activation function to the hidden layer

pass activated data from hidden layer to the output layer

apply chosen activation function to the output layer

END

***PROCESS**_start of backward pass*

calculate error signal and delta term for output layer

calculate error signal and delta term for hidden layer

compute weight error derivatives and adjust weights between hidden and output layers

compute weight error derivatives and adjust weights between input and hidden layers

compute quantities for rmse, mean error and ANN training

END

END

***PROCESS**_calculate errors*

calculate rmse and mean errors

check for maximum rmse

plot rmse

save rmse to array

END

***IF** rmse < specific value*

***EXIT** (computation has achieved desired accuracy)*

END

END

END

***PROCESS**_depths*

plot original water depths

plot ANN estimates of water depths

END

***PROCESS**_print summary*

print rmse, mean error, weight matrices, weight changes, targets, ANN estimates

***SAVE** new data*

END

4.8.2 Evaluation of the number of neurons in the hidden layer

Initially, numerous simulations were performed with differing learning rates, momentum terms and number of hidden neurons to obtain some idea as to the magnitudes of the RMSE (cf. table 4.1). To reduce the computational workload at this stage, the simulations were conducted over a 600 Km length of river and 10000 iterations. An ebb tide regime was used i.e. a down stream velocity at the river end of 0.25 m/s and at the estuarine end, a forcing velocity of 0.5 m/s . The still water depth h for this training set was chosen to be 15 m, the average depth of the Thames at the Medway confluence. Since the spatial difference between the sections was 0.5 Km (cf. Chapter 3), this meant there were 1800 inputs (two velocities and one depth at alternating sections) into this first network with 599 simulated outputs of depths.

The values in **bold** (in table 4.1 that follows) represent RMSE plots that were either highly oscillatory (and hence unstable) in the first 4000 iterations but did converge at the end of 10000 iterations, or converged but only in the latter 3000 iterations. The worst 'offenders' seemed to be when the momentum term was about 0.4 (for certain learning rates) and displayed instability due to their highly oscillatory nature. This was a point that Haykin [43] alluded to. Pragmatically, it is necessary to appreciate the significance of the data in the table 4.1. These values relate to a range of [1.1, 5.3] mm variation in the surface of the water level.

RMSE results – Depth training

	3 hidden neurons	5 hidden neurons	7 hidden neurons	9 hidden neurons	11 hidden neurons	
Learning Rate	10,000 iterations	10,000 iterations	10,000 iterations	10,000 iterations	10,000 iterations	Momentum Term
0.05	0.0033	0.0017	0.0027	0.0032	0.0019	0.4
0.1	0.0020	0.0018	0.0017	0.0023	0.0025	0.4
0.15			0.0025			0.4
0.2	0.0019	0.0025	0.0015	0.0018	0.0019	0.4
0.25	0.0020		0.0012			0.4
0.3	0.0035	0.0020	0.0013	0.0022	0.0020	0.4
0.05	0.0053	0.0037	0.0042	0.0023	0.0029	0.5
0.1	0.0018	0.0014	0.0072	0.0022	0.0039	0.5
0.15			0.0039			0.5
0.2	0.0021	0.0027	0.0034	0.0021	0.0025	0.5
0.25			0.0037			0.5
0.3	0.0014	0.0012	0.0018	**0.0020**	**0.0017**	0.5
0.4	0.0014	0.0026	0.0039	0.0020	0.0018	0.5
0.5	0.0011	0.0020	0.0011	0.0014	0.0009	0.5
0.05	0.0014	0.0035	0.0064	0.0019	0.0026	0.6
0.1	0.0025	**0.0024**	0.0019	0.0016	0.0021	0.6
0.15			0.0025			0.6
0.2	**0.0019**	0.0015	0.0019	**0.0031**	**0.0019**	0.6
0.25	**0.0016**	0.0025	0.0031	**0.0058**	0.0011	0.6
0.3	**0.0023**	0.0013	0.0012	0.0018	0.0012	0.6
0.4	**0.0014**	0.0022	0.0012	0.0012	**0.0014**	0.6
0.5	0.0014	0.0010	0.0010	0.0012	0.0008	0.6
0.05	0.0033	0.0042	0.0031	0.0057	0.0043	0.7
0.1	0.0023	0.0023	0.0020	0.0034	0.0021	0.7
0.15	0.0020	0.0054	0.0019	**0.0025**	0.0019	0.7
0.2	0.0020	0.0019	0.0017	0.0016	**0.0017**	0.7
0.25	0.0012	0.0016	0.0019	0.0015	0.0012	0.7
0.3	0.0012	0.0017	0.0016	0.0012	0.0012	0.7
0.4	0.0012	**0.0011**	0.0014	**0.0013**	**0.0022**	0.7
0.5	0.0011	0.0010	0.0012	0.0012	0.0018	0.7
0.05	0.0016	0.0023	0.0038	0.0019	0.0022	0.8
0.1	0.0021	0.0018	0.0036	0.0020	0.0017	0.8
0.15			0.0022			0.8
0.2	**0.0020**	0.0018	0.0018	0.0016	**0.0016**	0.8
0.25	**0.0019**	**0.0021**	0.0017	**0.0019**	0.0019	0.8
0.3	0.0019	0.0016	0.0019	0.0012	**0.0017**	0.8
0.4	0.0014	0.0015	0.0015	0.0018	0.0016	0.8

Table 4.1: Evaluation of no. of hidden neurons

After training, with ebb tide data, the network was presented with unseen data of a flood tide nature. This was easily obtained by regenerating the depth data using a change from the estuarine velocity of 0.5 m/s to one of -0.5 m/s. Plots of different hidden layer architectures i.e. 3, 5, 7, 9 and 11 follow:

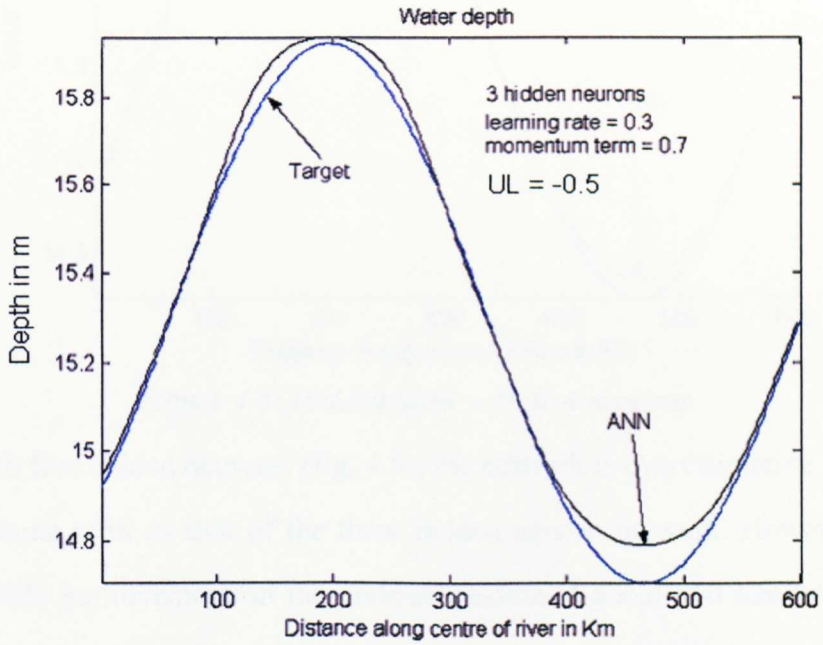

Figure 4.4: Hidden layer with three neurons

From the above figure, fig. 4.4, it can be seen that a hidden layer with three neurons is underestimating at 460 Km by about 0.1 m. It is correspondingly overestimating in the 200 Km region by about 0.025 m. The Admiralty have produced limited numerical hydrographic data of the Thames/Medway area. From Admiralty Chart no. 1185 (1997), used to produce the plan in Appendix F, it is observed that depths are quoted to the nearest 0.1 m and velocities to the nearest 0.1 knots. In other words, the Admiralty use accuracies of the order of +/- 0.05 m and +/- 0.0257 m/s. On this basis, the difference at 460 Km is definitely significant. Throughout the rest of this chapter (and also sections 5.1 to 5.4), where these differences are significantly greater than the measurement errors of the Admiralty data, then it will be argued that these differences are clearly not acceptable.

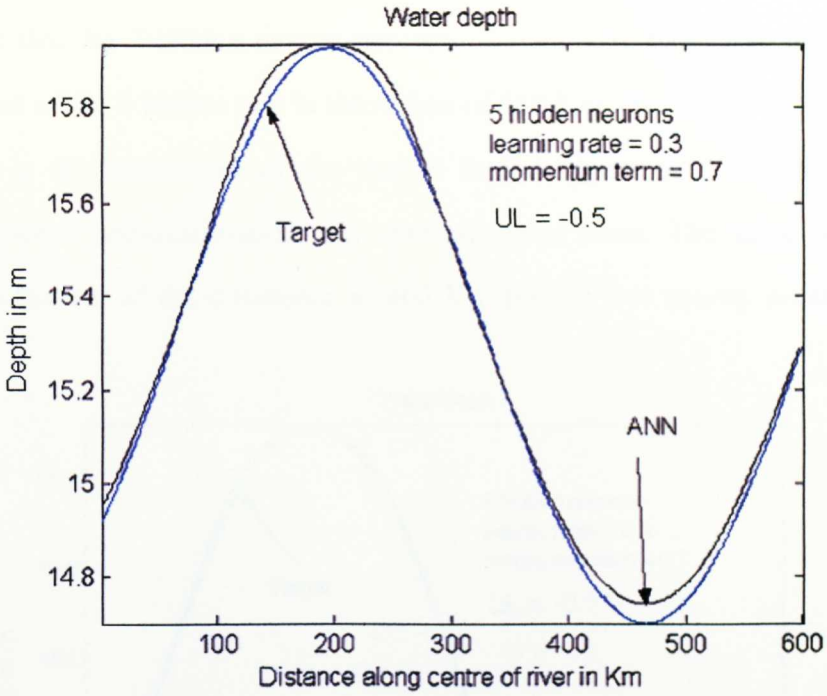

Figure 4.5: Hidden layer with five neurons

For a layer with five hidden neurons, (fig. 4.5), the network is overestimating in the 200 Km by about the same error as that of the three hidden neuron network. However there is an approximate 50% improvement on the underestimation at about 460 Km which is now of the order of 0.05 m. Also, this time the difference is not quite so significant at 460 Km being of the same order as that of the Admiralty value. It is still though, slightly larger than that obtained by Dibike (see Section 4.10.1).

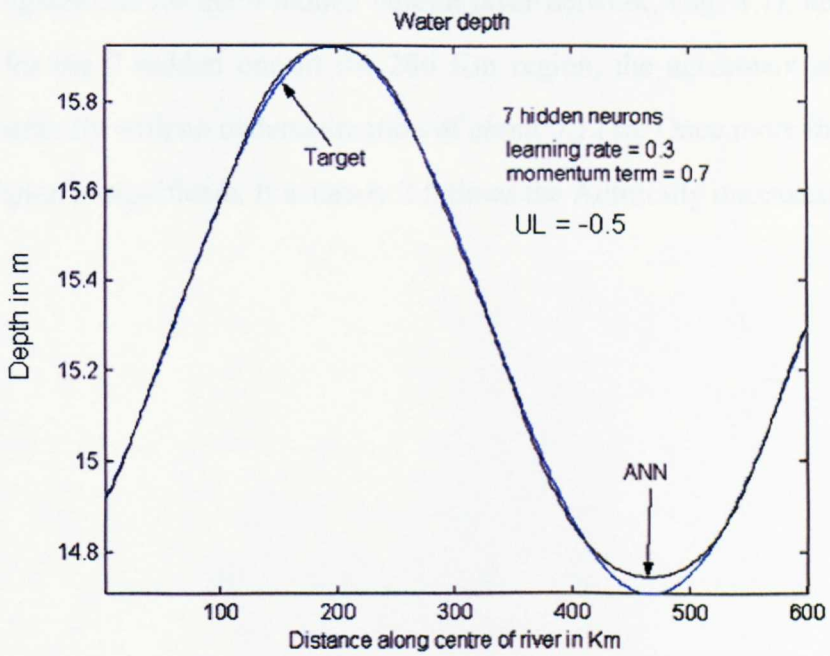

Figure 4.6: Hidden layer with seven neurons

It can be seen that the 7 hidden neuron network, (figure 4.6), has approximately the same accuracy as that of the 5 hidden one in the region of 460 Km. However, there is a significant improvement in the agreement of the curves in the region of 200 Km, with a slight oscillation between underestimation and overestimation there. The same comments with regard to significance of the difference at 460 Km for the five neuron network also apply here.

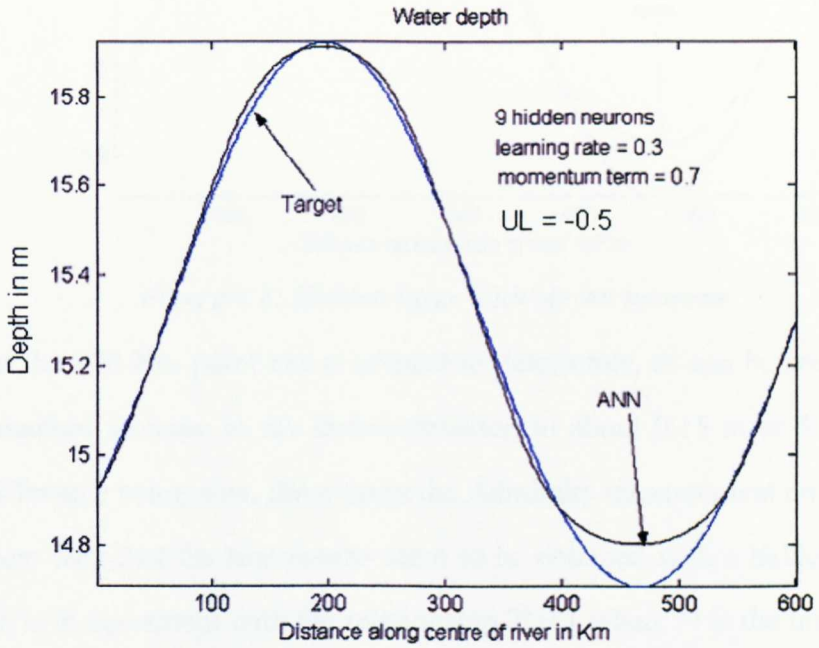

Figure 4.7: Hidden layer with nine neurons

Although the agreement for the 9 hidden neuron layer network, (fig. 4.7), appears to be as good as that for the 7 hidden one in the 200 Km region, the agreement at 460 Km has deteriorated markedly with an underestimation of about 0.12 m. Once more the difference at the 460 Km region is significant. It is nearly 2.5 times the Admiralty measurement error.

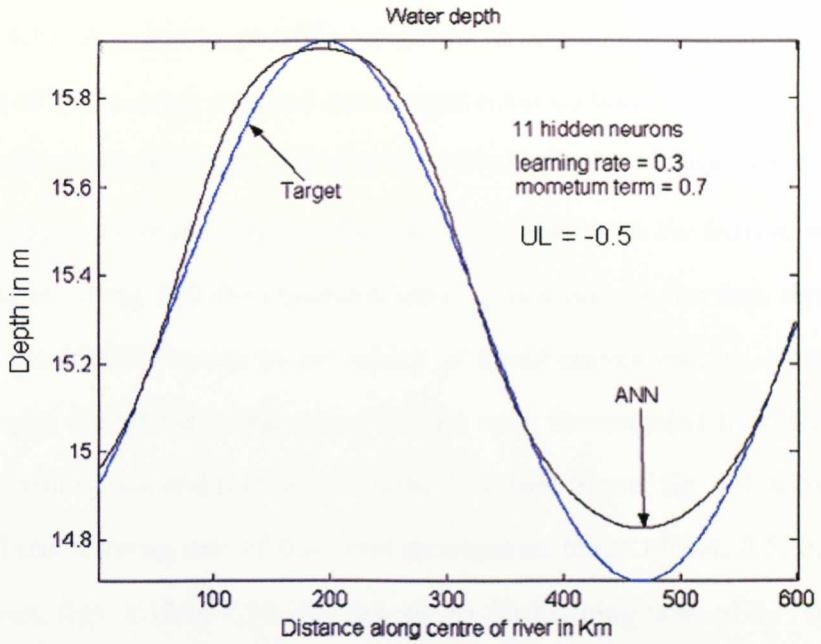

Figure 4.8: Hidden layer with eleven neurons

Agreement at the 200 Km point has continued to deteriorate, as can be seen from fig. 4.8, with also a marked increase in the underestimation to about 0.15 m at 460 Km. Again a significant difference being now, three times the Admiralty measurement error.

It would appear then that the best results seem to be obtained with a hidden layer of seven neurons. This is in agreement with the relationship 2N+1 where N is the number of units in the input layer, Marques de Sá [74].

4.8.3 Optimum learning rate and momentum parameters

It is however, fairly clear from table 4.1, that there is no significantly obvious indication of an optimum value for either the learning rate or momentum term. It was decided therefore to use the guidelines of Haykin [43] (pp. 193 – 197) to obtain data on these optimal values. To this end, the number of neurons in the hidden layer was maintained at seven and to avoid overfitting, a 1200 Km length of river was modelled. A model river length of 1200 Km with a spatial interval of 500m would provide 1199 data records not including the boundaries. Hence, by Sarle [92], this should ensure that the validation would not be very susceptible to 'overfitting', since with 36 weights in the network, 1080 records (30 x 36) would be required. Each of the 1199 generated records would contain two velocities and one depth

(cf. equation 4.1). A suitable possible guideline proposed by Haykin for obtaining the optimal values of the learning rate and momentum constant was:

'The η (learning rate) and α (momentum term) that on average yield convergence to a local minimum in the error surface of the network with the least number of epochs'

Essentially this is saying that the optimum learning rate and momentum term are obtained by inspecting the RMSE curves to see which of these curves reaches a convergent point first. Accordingly, the RMSE values per iteration were investigated for different parameter values of the learning rate and momentum term. The first figure, fig. 4.9, shows a plot of the RMSE for a fixed learning rate of 0.05 and momentum terms of 0.4, 0.5, 0.6, 0.7 and 0.8. The other figures, figs. 4.10 to 4.14, display results for learning rates of 0.1, 0.2, 0.3, 0.4 and 0.5 respectively with varying momentum terms:

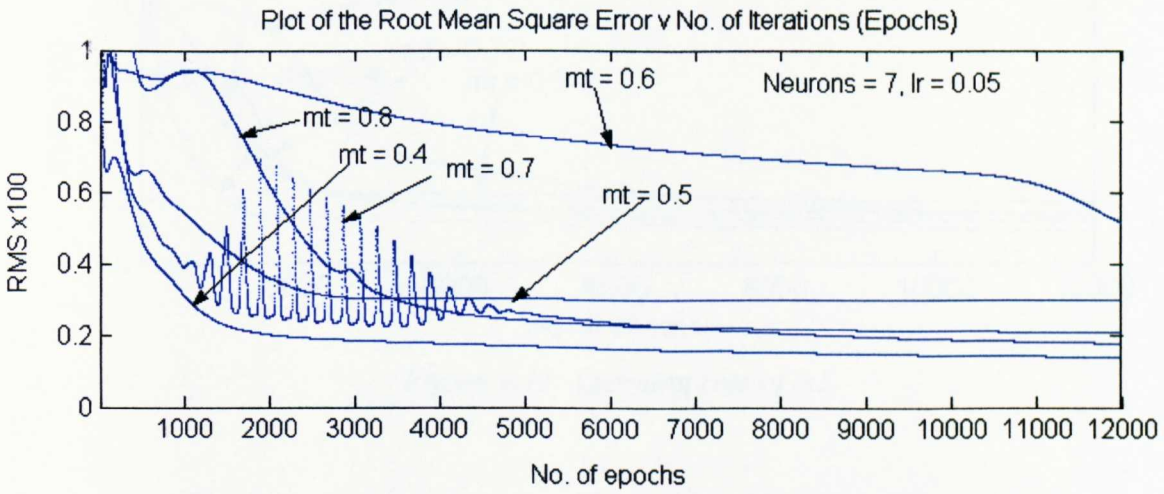

Figure 4.9: Learning rate of 0.05

It can be seen from the above figure that a momentum term of 0.4 gives a plot that converges more quickly than the others, converging to a minimum at about 2500 iterations.

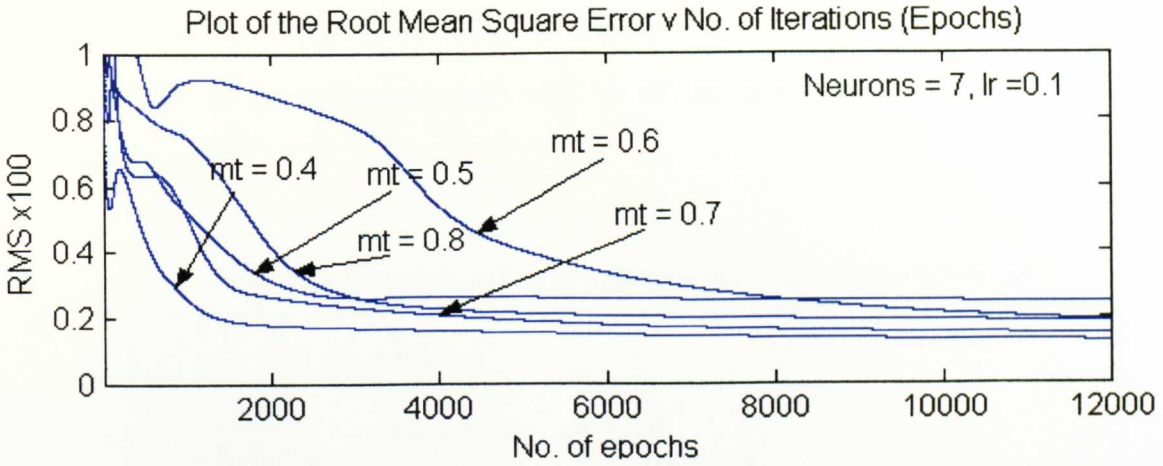

Figure 4.10: Learning rate of 0.1

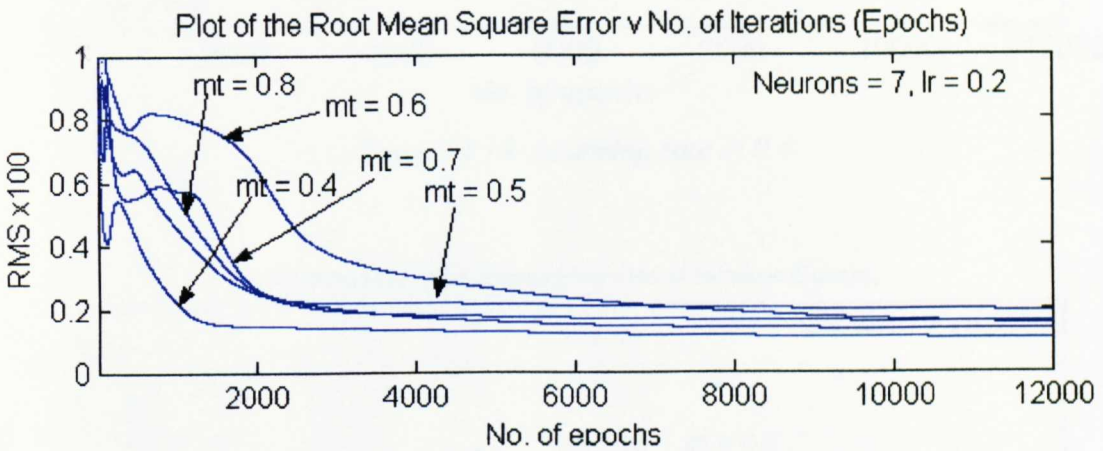

Figure 4.11: Learning rate of 0.2

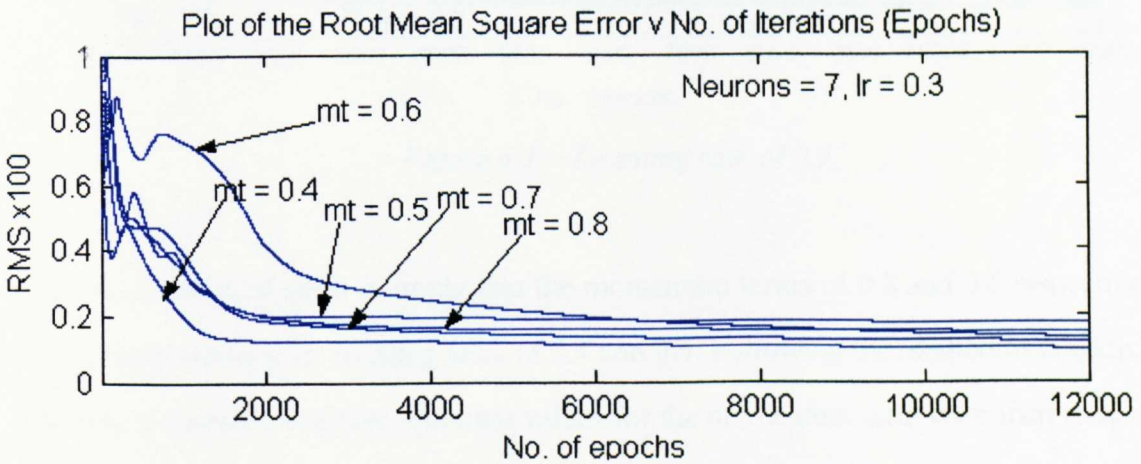

Figure 4.12: Learning rate of 0.3

Figs. 4.10 to 4.12, for learning rates of 0.1, 0.2 and 0.3, again indicate that a momentum term of 0.4 is the optimum value with all of the said curves converging before 2000 iterations.

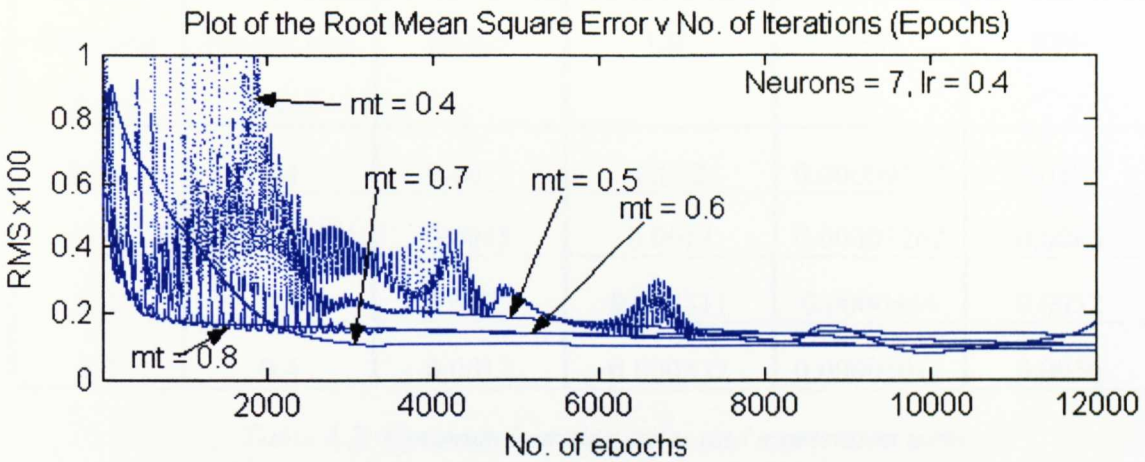

Figure 4.13: Learning rate of 0.4

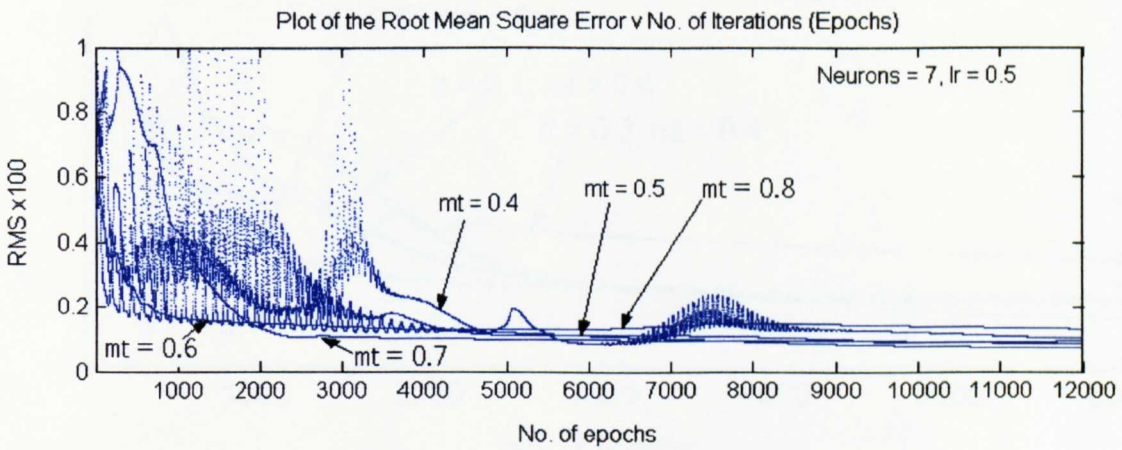

Figure 4.14: Learning rate of 0.5

Figs. 4.13 and 4.14 seem to imply that the momentum terms of 0.8 and 0.6 respectively are the optimum values for learning rates of 0.4 and 0.5. Following the method of Haykin, these learning rates and respective optimum values for the momentum term were then recalculated (over a shorter iteration period) and re-plotted. These plots were then used to finalize the choice of learning rate and momentum term. The learning rates / momentum terms of 0.4/0.6 and 0.5/0.6 proved to be highly oscillatory and unstable up until approximately

5,500 iterations and so were discarded. The finalized choices and their plots follow (conducted over 6000 iterations):

Optimum learning rate and momentum term parameters with associated errors					
Learning Rate	Momentum Term	RMSE	ME	ESD	RPE
0.05	0.4	0.0017	0.002	0.00009172	0.0129
0.1	0.4	0.0015	0.0013	0.00007262	0.0086
0.2	0.4	0.0025	0.000811	0.0000464	0.0052
0.3	0.4	0.0012	0.000872	0.00005015	0.0056

Table 4.2: Optimum learning rates and momentum terms

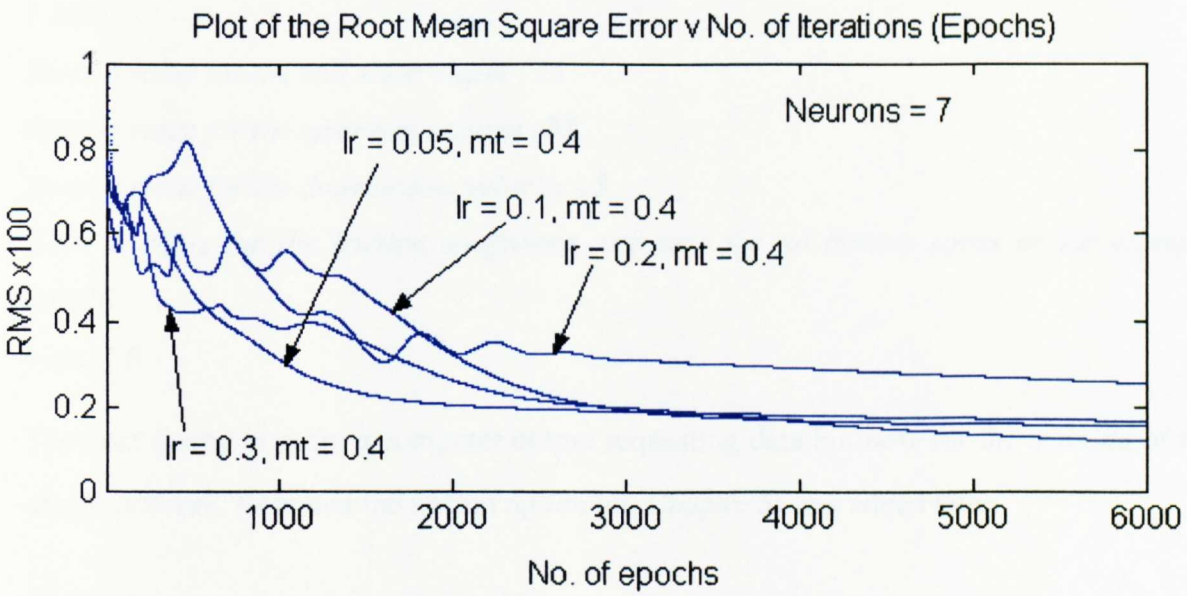

Figure 4.15: RMSE plots of optimum parameters

Following the arguments used in Section 4.6, the error ratios (using the values from table 4.2 above) are consistent with the values quoted there except for ME/RMSE relating to the first entry representing a learning rate of 0.05. This results in a ratio of 1.1765. In other words, contrary to the comment following equation **(4.26)**, although the ME is of a similar magnitude, it is now larger than the RMSE for this particular case. Although the scenario of a learning rate of 0.2 and momentum term of 0.4 appear to give the best error measure results overall, it is fairly obvious from fig. 4.15 that the optimum values are a learning rate

of 0.05 and a momentum term of 0.4 since it is this curve that converges first. These values are used throughout the rest of the research in this section on the depth analysis.

4.9 Simulation of the depth over 1200 Km

Having decided upon optimum values for the learning and momentum parameters, the depth was again modelled with 12000 iterations using the program (previously discussed) *DepthSolution* and an upstream velocity of 0.25 m/s and downstream (estuarine) ebb velocity of 0.5 m/s. The following 'italicized' listing is from the computer output requesting data to generate the depth profile (i.e. the target data) with user response in bold:

How many tidal cycles do you wish to iterate over (note 12.4 hours = one tidal cycle)? ***240***
tidal cycles = 240
Enter no. of sections required (must be odd). e.g., 101 spans 50 Km, 201 spans 100 Km etc.)
? ***2401***
Enter a value for the still water depth ***15***
Enter a value for the upstream velocity ***.25***
Enter a value for the downstream velocity ***-.5***
Enter a value for the friction coefficient, say zero for no bottom stress or for example 0.0026
Value? ***0***

The next listing is again a computer output requesting data but now for the *training* of the neural network. Note that the LeCun option (cf. Chapter 5) was added later:

This NN is designed to use two possible types of activation functions
1 - The sigmoid (logistic) function
2 - The tanh function
Your choice for the activation function i.e. 1 or 2 from above? ***2***
Choice for learning rate e.g. usually 0.1 to 1 ? ***.05***
Value for the momentum term e.g. usually 0.1 to 0.8 ? ***.4***
Required value for the root mean square error e.g. 0.01 ? ***.000000000005***
No. of neurons in the hidden layer e.g. 3 ? ***7***
How many iterations do you require ? ***12000***

The pattern of final and penultimate velocities and depths has now been loaded

Enlargement factor for rmse plot e.g. 10 to 100 ? ***100***

1 - Enter the weights manually ?

2 - Let the system automatically apply randomized ones ?

3 - Load saved weights for early stopping evaluation on a previously trained system ?

Your choice for the weights i.e. 1, 2 or 3 from above ? ***2***

Fig. 4.16 that follows is a plot of the 'raw' depth profile and the associated neural network simulation of it. The fit is extremely good and it is rather difficult to discern one graph from the other.

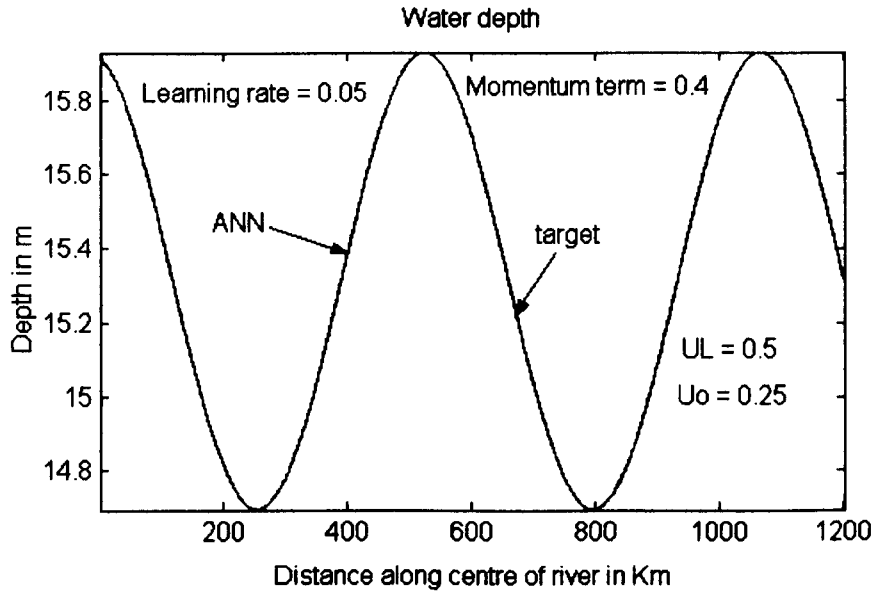

Figure 4.16:15 m depth ebb tide simulation

	0 Km	254 Km	526 Km	796 Km	1066 Km	1199 Km
ANN	15.9108	14.6921	15.9245	14.6906	15.9243	15.3216
Target	15.9134	14.6924	15.9271	14.6920	15.9270	15.3212
Difference	-0.0026	-0.0003	-0.0026	-0.0014	-0.0027	0.0004

Table 4.3: ANN and target variations

As the last figure and table 4.3 demonstrate, there is an extremely good agreement between the neural network output and the finite difference scheme data. All of the differences are within 3 mm. The following table contains an extract from the results – there are 1201 of

them including the boundaries, so only the first 20 are listed. Note that the figures in the body of the table represent deviations above or below the 15 m still water level

Calculated value(s) of the output layer		Target value(s) of the output layer
0.9108		0.9134
0.9096		0.9118
0.9084		0.9101
0.9072		0.9085
0.9059		0.9067
0.9045		0.9048
0.9029		0.9027
0.9013		0.9009
0.8996		0.8986
0.8977		0.8965
0.8957		0.8941
0.8936		0.8919
0.8914		0.8894
0.8891		0.8871
0.8866		0.8841
0.8841		0.8819
0.8814		0.8788
0.8787		0.8762
0.8757		0.8731
0.8728		0.8703

Table 4.4: ANN and target variations

The error measures, equations **(4.26)** to **(4.29)** during the course of the training were noted and were as follows:

Errors from training			
RMSE	ME	ESD	RPE
0.0014	0.00104272	0.000041074	0.0067

Table 4.5: Training errors

The error ratios (cf. Section 4.6) were respectively,

$$\frac{ME}{RMSE} \sim \pm 0.745, \quad \frac{ESD}{RMSE} \sim \pm 0.029 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.79$$

As can be seen from table 4.5, the 'fit' is extremely good with a mean absolute error of less than 2mm. After training had finished, the following finalized table of weights were obtained (tables 4.6a to 4.6f), the original values at the start of training also included for comparison:

Note that these tables need to be read in conjunction with fig. 4.1.

-0.6954	0.0138	0.9624	0.4104	-0.4838	-0.063	-0.1689
-1.1591	-1.5399	0.168	-0.699	1.0089	0.043	1.2015
0.6334	0.8051	0.5357	0.2734	-1.2898	-0.982	2.5339

Table 4.6a Final weights from input layer to hidden layer

-0.4326	0.2877	1.1892	0.1746	-0.5883	0.1139	-0.0956
-1.6656	-1.1465	-0.0376	-0.1867	2.1832	1.0668	-0.8323
0.1253	1.1909	0.3273	0.7258	-0.1364	0.0593	0.2944

Table 4.6b Original weights from input layer to hidden layer

-0.2629	-0.2739	-0.2268	0.2358	0.1045	-0.177	-0.0733
0.5065	-0.3935	0.2056	-0.5123	-1.1742	-1.0237	2.0338
0.5081	-0.3859	0.2084	-0.4524	-1.1534	-1.0413	2.2395

Table 4.6c Change from original

Final weight matrix from Bias unit(s) to hidden layer:		Original weight matrix from Bias unit(s) to hidden layer:		Change from original:
-1.3613		-1.441		0.0796
0.4138		0.5711		-0.1574
-0.406		-0.3999		-0.0061
0.5709		0.69		-0.1191
0.5846		0.8156		-0.231
0.4392		0.7119		-0.2727
1.8361		1.2902		0.5458

Table 4.6d Weight matrix from Bias unit to hidden layer

Final weight matrix from hidden layer to output layer:		Original weight matrix from hidden layer to output layer:		Change from original:
-1.4809		-1.3362		-0.1448
0.3651		0.7143		-0.3492
0.2931		1.6236		-1.3304
0.524		-0.6918		1.2157
-0.1231		0.858		-0.9811
0.3632		1.254		-0.8908
-2.1884		-1.5937		-0.5947

Table 4.6e Final weights from hidden layer to output layer

Final weight matrix from Bias unit(s) to output layer:		Original weight matrix from Bias unit(s) to output layer:		Change from original:
0.6952		0.6686		0.0266

Table 4.6f Weight matrix from Bias unit to output layer

It is interesting to note that (tables 4.6a and 4.6b) just under half of the weights have been reduced in size (ignoring sign) by the backpropagation algorithm whilst the remainder have been magnified in size. The average magnification factor was 2.98, the smallest being 0.048 and the largest, 16.56. The changes to the bias weights to the hidden layer (table 4.6d) were much less dramatic. No changes of sign occurred and the average magnification factor was 0.896. Considering table 4.6e, it can be seen that there were some dramatic changes to the weights from the hidden layer to the output layer. Two of the weights changed sign whilst the average magnification factor was 0.623. For the last connection weight, table 4.6f, from the bias unit to the output unit, the change was minimal in comparison. It seems then that the major changes have been to the weights from the input layer to the hidden layer and then to a lesser degree, that from the hidden layer to the output layer.

4.10 Validation of the depth simulation

Validation was accomplished using a large data file (again 1201 records) to ensure limited overfitting as previously mentioned. The data, again as previously mentioned whilst discussing the initial investigations over a 600 Km, was easily obtained by changing the downstream parameter from an ebb tide regime to that of a flood tide one. Hence for the

validation, the parameters were an: upstream velocity of 0.25 m/s, downstream velocity of -0.5 m/s, still water depth of 15 m, river length of 1200 Km, learning rate of 0.05 and a momentum term of 0.4. The data file for this flood tide was created using the *VenantGenerator2* program again and then to test the network, a program entitled *DepthSolutionValidation* was used to verify the results. The program of course uses the (fixed) weights from the training session. This program is listed in appendix C.

4.10.1 Pseudo Code of the depth validation program

A listing of the pseudo code is appended below:

Filename: DepthSolutionValidation

***COMMENT**:This program is used to validate the weights and solution of the DepthSolution ANN program. It uses the weights created by the DepthSolution program stored in a data file DepthValidation. It also requires for validation purposes, unseen data created and saved using another different run of the file VenantDataGenerator2.*

***PROCESS**_get data and initialize arrays*

***LOAD** file containing original hydraulic data computed using VenantDataGenerator2*

***LOAD** file containing ANN simulation of the depths and related weights*

***INITIALIZE** pattern, input layer and output layer arrays*

***INITIALIZE** arrays to hold weights*

***INITIALIZE** variables bias, iteration counter, mean error, max rmse*

LOOP

fill target matrix with data from file

fill pattern matrix with data from file

END

END

PROCESS

***LOOP**_ iteration counter*

select input pattern

select target pattern

***PROCESS**_start of forward pass*

```
                    pass weighted input data to the hidden layer
                    apply chosen activation function to the hidden layer
                    pass activated data from hidden layer to the output layer
                    apply chosen activation function to the output layer
                END
                PROCESS
                    calculate rmse and mean errors
                    check for maximum rmse
                    plot rmse
                    save rmse to array
                END
        END
END
PROCESS_depths
        plot original water depths
        plot ANN estimates of water depths
END
PROCESS_print summary
        print rmse, mean error, weight matrices, weight changes, targets, ANN estimates
        SAVE new data
END
```

Fig. 4.17 that follows is a plot of the 'raw' depth profile of the flood tide regime and the associated neural network simulation of it. As with the ebb tide regime (fig. 4.16), it is again an extremely good 'fit' and it is rather difficult to discern one graph from the other.

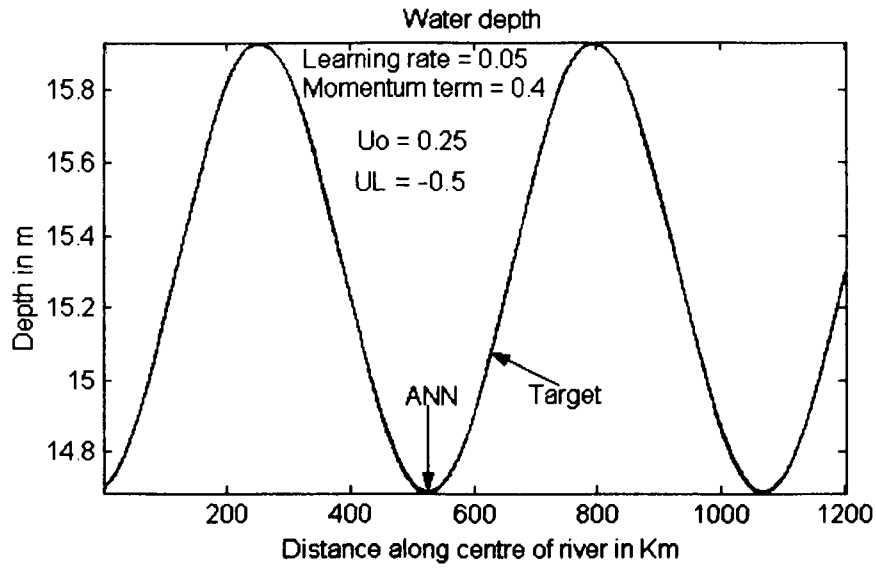

Figure 4.17: 15 m depth flood tide simulation

	0 Km	253 Km	526 Km	793 Km	1066 Km	1199 Km
ANN	14.6972	15.9211	14.6854	15.9211	14.6853	15.2933
Target	14.7045	15.9274	14.6923	15.9272	14.6922	15.2944
Difference	-0.0073	-0.0063	-0.0069	-0.0061	-0.0069	-0.0011

Table 4.7 ANN and target variations

Table 4.7 provides a clearer indication of the agreement between the two curves at different points along the river length. The neural network is underestimating the target data but still within a very acceptable 8 mm.

Errors from validation			
RMSE	ME	ESD	RPE
0.0065	0.0043	0.00028695	0.0360

Table 4.8: Validation errors

Comparing the errors in the validation of the network (table 4.8) with that of the training (table 4.5); the RMSE, ME, ESD and RPE have been 'magnified' by approximately 4.6, 4.1, 7 and 5.4 respectively. Dibike [31] in a slightly similar model (with a little bed slope), obtained an RMSE of 0.0048 and by estimation from the graphs in his paper, a mean error of 0.04 m.

The error ratios are now:

$$\frac{ME}{RMSE} \sim \pm 0.66, \quad \frac{ESD}{RMSE} \sim \pm 0.044 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.54$$

Finally, after validating the network, the following table of weights is obtained. This table needs to be read in conjunction with fig. 4.1:

	Final weight matrix from input layer to hidden layer:						
	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
X_0	-1.3613	0.4138	-0.4060	0.5709	0.5846	0.4392	1.8361
X_1	-0.6954	0.0138	0.9624	0.4104	-0.4838	-0.063	-0.1689
X_2	-1.1591	-1.5399	0.168	-0.699	1.0089	0.043	1.2015
X_3	0.6334	0.8051	0.5357	0.2734	-1.2898	-0.982	2.5339

	Final weight matrix from hidden layer to output layer:							
	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
Z	0.6952	-1.4809	0.3651	0.2931	0.5240	-0.1231	0.3632	-2.1884

Table 4.9: Final weight matrices

4.11 Training the velocity network (zero bottom friction)

The methodology of the velocity training and validation is similar to that of the depth (fig. 4.3) and so will not be repeated here. In addition, the pseudo code of the *VelocitySolution* program is virtually the same in structure as that of the *DepthSolution* program and so has been omitted.

The Matlab program to perform the training is *VelocitySolution* and is listed in appendix C. As for the analysis of the depth simulations, some initial investigations were made with regard to the RMSE values during velocity simulations over a period of 10000 iterations. The table below lists the results:

RMSE – velocity training

	3 hidden neurons	5 hidden neurons	7 hidden neurons	9 hidden neurons	11 hidden neurons	
Learning Rate	10,000 iterations	10,000 iterations	10,000 iterations	10,000 iterations	10,000 iterations	Momentum Term
0.05	0.0013	0.0012	0.0018	0.0022	0.002	0.4
0.1	0.002	0.0014	0.0011	0.0013	0.0019	0.4
0.2	0.0013	0.0023	0.0011	0.0008	0.0009	0.4
0.3	0.0013	0.002	0.0007	0.0028	0.001	0.4
0.4	0.0009	0.0013	0.0013	0.0019	0.0013	0.4
0.05	0.0014	0.0022	0.0023	0.0023	0.002	0.5
0.1	0.0011	0.0011	0.0012	0.0017	0.0019	0.5
0.2	0.0014	0.0017	0.0016	0.0011	0.0015	0.5
0.3	0.0014	0.0008	0.0011	0.0014	0.0018	0.5
0.4	0.0012	0.0016	0.0016	0.002	0.0019	0.5
0.05	0.0024	0.002	0.0011	0.0044	0.0026	0.6
0.1	0.0015	0.0022	0.0027	0.0019	0.0011	0.6
0.2	0.002	0.0014	0.0025	0.0026	0.0017	0.6
0.3	0.0019	0.0019	0.0026	0.0019	0.0012	0.6
0.4	0.0011	0.0013	0.0006	0.0016	0.0014	0.6
0.05	0.0033	0.0026	0.0009	0.0017	0.0047	0.7
0.1	0.0024	0.0013	0.0012	0.0015	0.0061	0.7
0.2	0.0029	0.0009	0.0013	0.0012	0.0038	0.7
0.3	0.0012	0.0017	0.0008	0.0027	0.0007	0.7
0.4	0.0015	0.0056	0.0011	0.0023	0.0013	0.7
0.05	0.0021	0.0013	0.0023	0.0031	0.0021	0.8
0.1	0.0016	0.0028	0.0023	0.0134	0.0016	0.8
0.2	0.0023	0.001	0.0016	0.0012	0.0018	0.8
0.3	0.0022	0.0012	0.0009	0.0016	0.0035	0.8
0.4	0.0014	0.0016	0.0018	0.0013	0.0018	0.8

Table 4.10: Evaluation of no. of hidden neurons

By inspection of the table above, it was noted that the lowest RMSE average occurred for a hidden layer with seven neurons which in itself was desirable in order to be consistent with the architecture of the depth network. The RMSE averages for 3, 5, 7, 9 and 11 neurons were respectively 0.0017, 0.0018, 0.0015, 0.0024 and 0.0021. However, as before with the depth simulations, suitable choices for the learning rate and momentum parameters were not obvious. Consequently, the guideline (cf. Section 4.8.3) promoted by Haykin as used previously, was applied again.

4.11.1 Optimum learning rate and momentum term parameters

After running the simulations for the learning rates of 0.05, 0.1, 0.2, 0.3, 0.4 and 0.5 with associated momentum values, the four best combinations that were obtained (simulated over 8000 iterations) are listed in the table below followed by a plot of their RMSE graphs. The graph of the learning rate / momentum term of 0.5/0.4 was discarded as it was highly oscillatory until about 6000 iterations.

Optimum learning rate and momentum term parameters with associated errors					
Learning Rate	Momentum Term	RMSE	ME	ESD	RPE
0.05	0.4	0.002	0.001495619	0.0000569	0.0078000
0.1	0.4	0.0026	0.001720766	0.0000740	0.0113000
0.2	0.7	0.0021	0.001177212	0.0000615	0.0085000
0.3	0.4	0.0024	0.001333333	0.0000698	0.0094000
0.4	0.5	0.0024	0.001531957	0.0000696	0.0097000

Table 4.11: Optimum learning rates and momentum terms

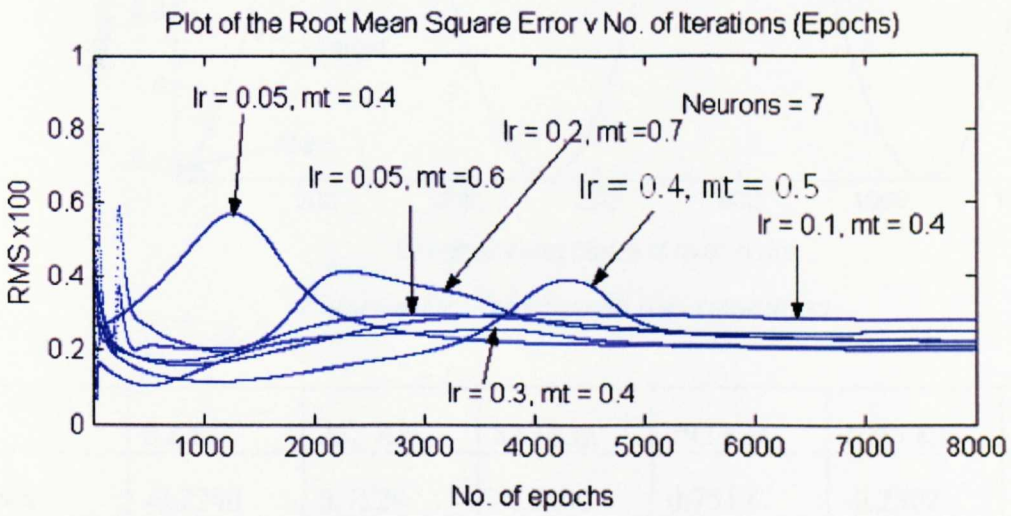

Figure 4.18: RMSE plots of optimum parameters

From table 4.11, the average error ratios are:

$$\frac{ME}{RMSE} \sim \pm 0.63, \quad \frac{ESD}{RMSE} \sim \pm 0.029 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.05$$

By inspection of the fig. 4.18, it can be seen that the RMSE graph with a learning rate of 0.05 and momentum term of 0.4 converges before any of the others at approximately 3,500 iterations. It also has the smallest RMSE error. On this basis, it was decided to choose optimum values for the learning rate and momentum term to be 0.05 and 0.4 respectively as before for both the depth and velocity simulations.

4.12 Simulation of the velocity over 1200 Km.

Using these optimum values for the learning and momentum parameters i.e. 0.05 and 0.4 respectively, the velocity was modelled using the program (previously discussed) *VelocitySolution* with an upstream velocity of 0.25 m/s and a downstream (estuarine) ebb tide velocity of 0.5 m/s. Fig. 4.19 that follows is a plot of the velocity profile and the associated neural network simulation of it. As with the depth profiles, the 'fit' is extremely good and it is rather difficult to discern one graph from the other.

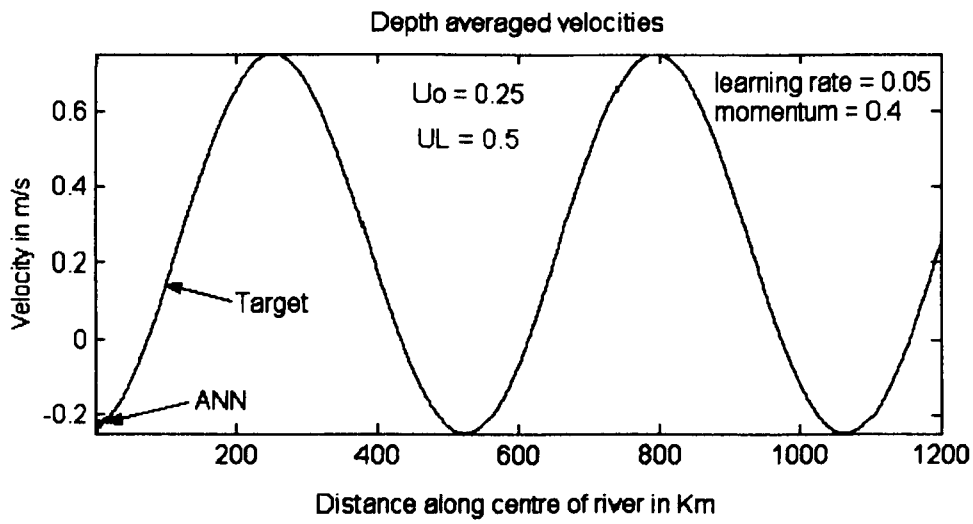

Figure 4.19: Velocity ebb tide simulation

	0 Km	252 Km	525 Km	793 Km	1065 Km	1200 Km
ANN	-0.2240	0.7520	-0.2500	0.7517	-0.2502	0.2467
Target	-0.2394	0.7520	-0.2500	0.7519	-0.2501	0.2477
Difference	-0.0154	0.0000	0.0000	0.0002	0.0001	0.0010

Table 4.12: ANN and target variations

Further, it can be seen from table 4.12, that this 'fit' is supported by the extremely good agreement between the finite difference scheme data and the neural network output. The largest discrepancy is less than 16 mm. The following table contains an extract from the results –as for the depth results, there are 1201 of them, and so only, the first 20 are listed. The figures in the body of the table represent actual velocities (not deviations) in m/s.

Calculated value(s) of the output layer		Target value(s) of the output layer
-0.2240		-0.2394
-0.2252		-0.2382
-0.2399		-0.2368
-0.2500		-0.2356
-0.2442		-0.2341
-0.2292		-0.2327
-0.2200		-0.2310
-0.2238		-0.2296
-0.2332		-0.2278
-0.2363		-0.2262
-0.2287		-0.2242
-0.2182		-0.2225
-0.2136		-0.2204
-0.2171		-0.2187
-0.2217		-0.2164
-0.2209		-0.2144
-0.2136		-0.2122
-0.2063		-0.2100
-0.2039		-0.2076
-0.2060		-0.2053

Table 4.13: ANN and target variations

During the course of training the velocity network, the error measures **(4.26)** to **(4.29)** were noted and were as follows:

Errors from training			
RMSE	ME	ESD	RPE
0.0011	0.0006179	0.0000328	0.0035

Table 4.14: Training errors

giving error ratios of:

$$\frac{ME}{RMSE} \sim \pm 0.56, \quad \frac{ESD}{RMSE} \sim \pm 0.030 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 3.18$$

Although different quantities (that is, depth and velocity) are being compared here, it is interesting to note, from tables 4.5 and 4.14, that the ratios of velocity RMSE / depth RMSE and velocity ESD/depth ESD are both virtually the same with a value of 0.79. In other words, these two errors in the depth training have been reduced by about 20% in the velocity training. The ME for the velocity is about 60% of the corresponding value for that of the depth and similarly, the RPE for the velocity is about 52% of that for the depth.

After training had finished, the following finalized table of weights were obtained (tables 4.15a to 4.15f), the original values at the start of training also included for comparison. As for the depth training, these tables need to be read in conjunction with fig. 4.1:

-1.0205	-0.4243	1.4289	0.6306	0.1276	-0.8628	0.4027
-0.9043	1.4488	-0.7761	0.4766	0.5726	0.2918	0.4835
-1.3480	1.4992	0.1519	-1.7906	-0.8769	-0.7283	-0.4055

Table 4.15a Final weights from input layer to hidden layer

-0.3775	-0.2340	1.4435	0.7990	0.2120	-0.7420	0.3899
-0.2959	0.1184	-0.3510	0.9409	0.2379	1.0823	0.0880
-1.4751	0.3148	0.6232	-0.9921	-1.0078	-0.1315	-0.6355

Table 4.15b Original weights from input layer to hidden layer:

-0.6430	-0.1903	-0.0146	-0.1685	-0.0844	-0.1208	0.0128
-0.6084	1.3303	-0.4251	-0.4643	0.3348	-0.7905	0.3955
0.1271	1.1844	-0.4713	-0.7985	0.1309	-0.5968	0.2300

Table 4.15c Change from original:

Final weight matrix from Bias unit(s) to hidden layer:		Original weight matrix from Bias unit(s) to hidden layer:		Change from original:
-1.0827		-1.1878		0.1051
-2.0203		-2.2023		0.182
0.8822		0.9863		-0.1041
-0.9457		-0.5186		-0.4271
0.3006		0.3274		-0.0268
0.1823		0.2341		-0.0517
0.0238		0.0215		0.0023

Table 4.15d: Weight matrix from Bias unit to hidden layer

Final weight matrix from hidden layer to output layer:		Original weight matrix from hidden layer to output layer:		Change from original:
-1.9955		-0.5596		-1.4359
-0.2137		0.4437		-0.6574
-0.0025		-0.9499		0.9474
1.2238		0.7812		0.4426
0.1276		0.569		-0.4414
0.0324		-0.8217		0.8541
0.0977		-0.2656		0.3633

Table 4.15e: Final weights from hidden layer to output layer

Final weight matrix from Bias unit(s) to output layer:		Original weight matrix from Bias unit(s) to output layer:		Change from original:
-0.8717		-1.0039		0.1323

Table 4.15f: Weight matrix from Bias unit to output layer

Inspecting the weights (as was done for the depth simulation), it can be seen that the weights between the input layer and the hidden layer (tables 4.15a and 4.15b) have been magnified, ignoring sign, by the backpropagation by approximately on average, a factor of 2.38. The weights from the bias unit to the hidden layer showed little evidence of change, there being no sign changes and a magnification factor of just 1.050. Of the hidden to output layers weights there were three sign changes with on average, a multiplication factor of 0.895. The largest factor was 3.57 and the smallest, 0.003. Finally, the bias to output displayed just

moderate changes with a magnification factor of 0.868. Comparing this with the results for the depth, it is noted that the same pattern is being repeated here. The magnification factor acting on the weights between the input layers and hidden layers is about two to three times that of the factors between the other units and further, individual changes to the weights are sometimes much more dramatic.

4.13 Validation of the velocity network

In similar fashion to the method adopted for the depth network validation, a large data file (1201 records) was employed to ensure limited overfitting. The data was easily obtained by changing the downstream parameter from an ebb tide regime to that of a flood tide one. Hence for the validation, the parameters were an: upstream velocity of 0.25 m/s, downstream velocity of -0.5 m/s, still water depth of 15 m, river length of 1200 Km, learning rate of 0.05 and a momentum term of 0.4. The data file for this flood tide was created using the *VenantGenerator2* program again (as for the depth) and then to test the network, a program entitled *VelocityDepthSolutionValidation* was used to verify the results. The program of course uses the (fixed) weights from the training session. This program is listed in appendix C. The pseudo code of the *VelocitySolutionValidation* program is almost identical in structure to that of the *DepthSolutionValidation* program and so has been omitted.

Fig. 4.20 that follows is a plot of the velocity profile from the finite difference scheme of the flood tide regime and the associated neural network simulation of it. As with the ebb tide regime (fig. 4.21), it is again an extremely good 'fit' with just discernible discrepancies at 250 Km and 800 Km.

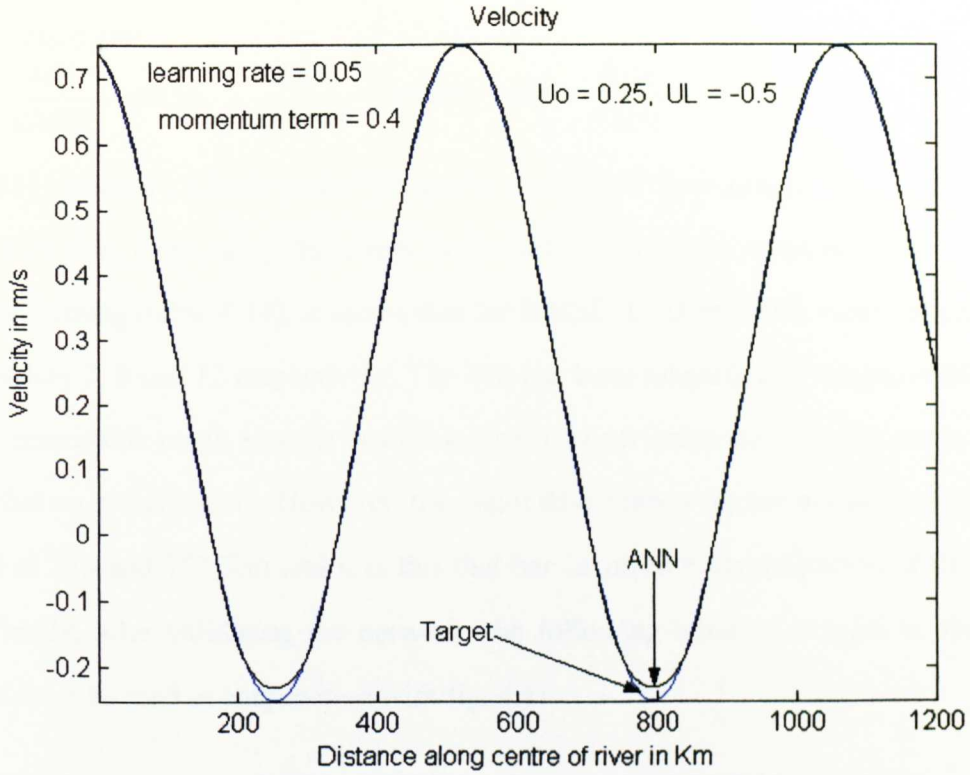

Figure 4.20: Flood tide simulation

	0 Km	253 Km	523 Km	793 Km	1065 Km	1200 Km
ANN	0.7390	-0.2306	0.7485	-0.2306	0.7484	0.2509
Target	0.7397	-0.2501	0.7518	-0.2500	0.7518	0.2502
Difference	0.0007	-0.0195	0.0033	-0.0194	0.0034	-0.0007

Table 4.16: ANN and target variations

From table 4.16, which depicts the agreement between the two curves at different points along the river length, it can be concluded that the neural network is overestimating the target velocities at 253 Km and 793 Km. However, all of the errors in estimation are within a very acceptable 20 mm/s

Errors from validation			
RMSE	ME	ESD	RPE
0.0079	0.006723	0.000264	0.0433

Table 4.17: Validation errors

The error ratios are:

$$\frac{ME}{RMSE} \sim \pm 0.56, \quad \frac{ESD}{RMSE} \sim \pm 0.030 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 3.18$$

Dibike [31] obtained a RMSE of 0.0085 and by estimation from graphs in his paper, a mean error of 0.05 m/s. Comparing the errors in the validation of the network (table 4.17) with that of the training (table 4.14), it seems that the RMSE, ESD and RPE were 'magnified' by approximately 7, 8 and 12 respectively. The ME has been magnified by approximately 10.8. This is a reasonable result since it is well known that performance on a test set is never as good as that on a training set. However, the major discrepancy (as has already been noted) is localized at 253 and 793 Km and it is this that has incurred a magnification of the training errors. Finally, after validating the network, the following table of weights is obtained, a table that must be read in conjunction with fig. 4.1:

Final weight matrix from input layer to hidden layer:							
	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
X_0	-1.0827	-2.0203	0.8822	-0.9457	0.3006	0.1823	0.0238
X_1	-1.0205	-0.4243	1.4289	0.6306	0.1276	-0.8628	0.4027
X_2	-0.9043	1.4488	-0.7761	0.4766	0.5726	0.2918	0.4835
X_3	-1.3480	1.4992	0.1519	-1.7906	-0.8769	-0.7283	-0.4055

Final weight matrix from hidden layer to output layer:								
	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
Z	-0.8717	-1.9955	-0.2137	-0.0025	1.2238	0.1276	0.0324	0.0977

Table 4.18: Final weight matrices

4.14 Comparison with other models

It would be useful at this stage to compare the model for accuracy with a model totally unrelated to neural networks before proceeding to the primary aim, a neural network incorporating bed stress. The neural network models in this chapter, that is, the depth model as given by fig. 4.1 and table 4.9 and the velocity as indicated by fig. 4.1 and table 4.18, were compared with a model (coded in FORTRAN) of Johns [51]. It is a model based purely on the basic hydrodynamical equations of momentum and continuity and an appropriate finite difference scheme with a zero Coriolis parameter. His model does not allow for bed friction and represents propagation of a Kelvin wave through a wide channel.

However, with the Coriolis parameter set to zero, the model assumes the degenerative form representing a simple shallow water wave in a one-dimensional channel. It is in this form that his model is used in the comparison with the neural network.

John's model was modified slightly to run over a length of 1200 Km. This was necessary since the neural network needed to be validated over that length for reasons already stated. The model was further modified to work on 1488 time steps in a 12.4 hour tidal period rather than 1200 over a 12 hour tidal period. In the tables that follow, tables 4.19 to 4.22, only the first ten results are shown that is, the first 10 Km. However, EXCEL plots (figs. 4.21 to 4.24) of these comparisons are included to indicate the degree of agreement over the full 1200 Km.

Two comparisons were made, the first using the training results (tables 4.19 and 4.20) and the second, using the results of the validation with unseen data (tables 4.21 and 4.22). The training was performed using a learning rate of 0.05, a momentum term of 0.4 and a hidden layer with seven units, as usual. The still water depth was 15 m as before.

4.14.1 Comparison with the training results

The training data was created using an imposed constant input river velocity at the upstream end of 0.25 m/s, whilst the corresponding quantity at the estuarine end was 0.5 m/s (signifying an outgoing ebb tidal flow). It should be noted that in the FORTRAN model, the estuarine value that should be entered is -0.618274208, since that model is defined by the wave amplitude there. This value is simply $U_L \times \sqrt{h/g}$.

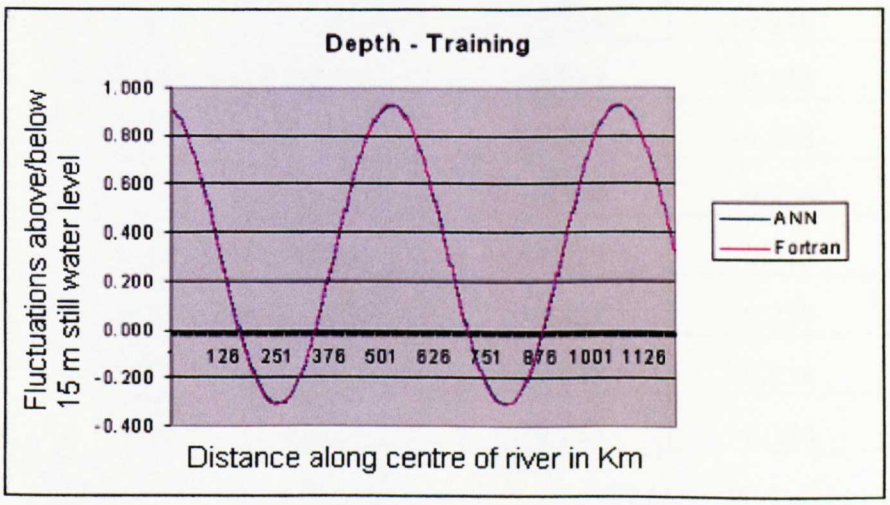

Figure 4.21: Ebb tide depth comparison

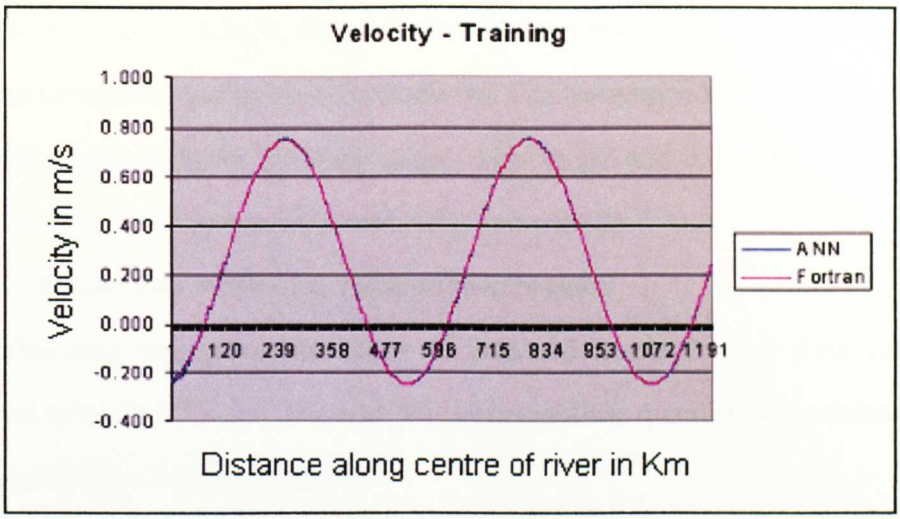

Figure 4.22: Ebb tide velocity comparison

In table 4.19 that follows, the first two columns are the ***fluctuations*** above or below the 15 m still water level, the first being the Fortran model and the second, the neural network model. The data relates to the first ten Km points at one Km intervals. The third column lists the absolute differences between these columns. The average and maximum absolute differences for all 1201 values (column 3) were 0.007 and 0.013 m respectively.

Fortran Model Depth	ANN Model Depth	Absolute value of difference
0.912	0.912	0.001
0.910	0.911	0.001
0.908	0.910	0.002
0.906	0.909	0.002
0.905	0.907	0.003
0.902	0.906	0.004
0.900	0.904	0.004
0.898	0.903	0.005
0.896	0.901	0.005
0.894	0.899	0.005

Table 4.19 (Depth network)

Fortran Model Velocity	ANN Model Velocity	Absolute value of difference
-0.237	-0.232	0.005
-0.236	-0.233	0.003
-0.235	-0.240	0.006
-0.233	-0.245	0.011
-0.232	-0.238	0.006
-0.230	-0.226	0.004
-0.229	-0.222	0.006
-0.227	-0.230	0.003
-0.225	-0.238	0.013
-0.223	-0.234	0.011

Table 4.20 (Velocity network)

The description of table 4.20 is the same as that for table 4.19 other than that it refers to actual values of velocity rather than fluctuations. The corresponding average and maximum absolute differences again for all 1201 results were 0.002 and 0.021 m/s. In practical terms then, there is very good agreement between the two models.

4.14.2 Comparison with the validation results

The validation data was generated using an imposed constant input river velocity at the upstream end of again 0.25 m/s, but now the corresponding quantity at the estuarine end was -0.5 m/s (signifying a flood tidal flow).

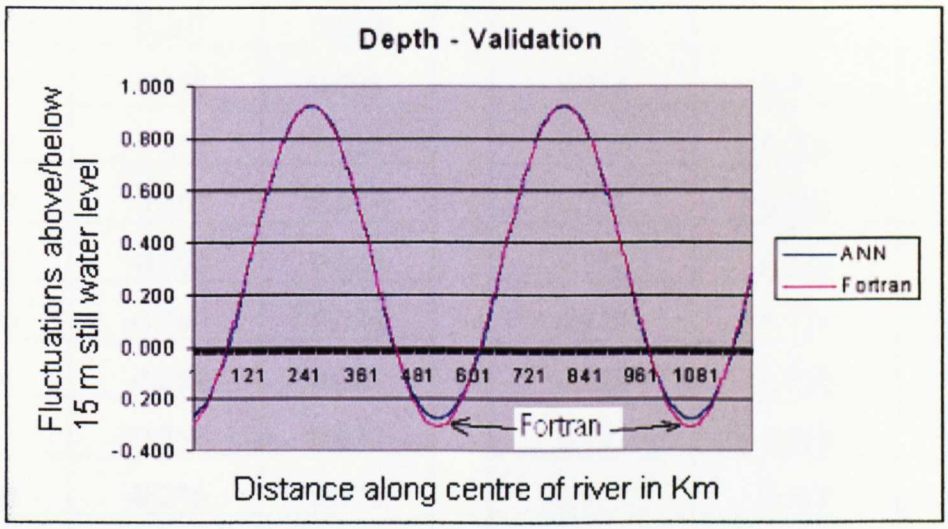

Figure 4.23: Flood tide depth comparison

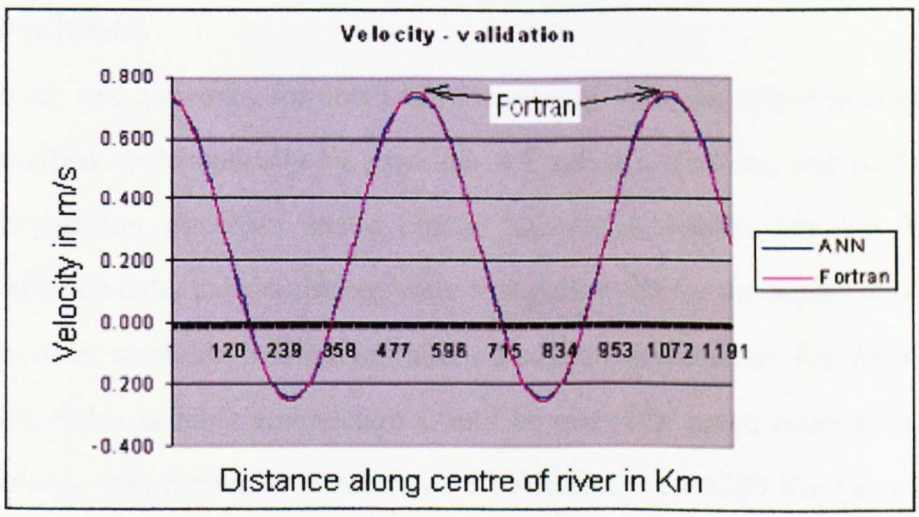

Figure 4.24: Flood tide velocity comparison

In table 4.21 that follows, the average and maximum absolute differences for the full 1200 Km were 0.012 and 0.033 m. Correspondingly, for the velocity (table 4.22), these differences were 0.008 and 0.023 m/s. As expected, the network is not quite as accurate in the validation as it is in the training, although again, in practical terms, the difference between the performance of training and validation here is insignificant.

Fortran Model Depth	ANN Model Depth	ABS Value of Diff.
-0.295	-0.267	0.028
-0.293	-0.266	0.028
-0.292	-0.265	0.027
-0.290	-0.263	0.027
-0.288	-0.262	0.026
-0.286	-0.260	0.026
-0.284	-0.259	0.025
-0.282	-0.257	0.025
-0.280	-0.256	0.024
-0.278	-0.254	0.024

Table 4.21 (Depth network)

Fortran Model Velocity	ANN Model Velocity	ABS Value of Diff.
0.737	0.726	0.011
0.736	0.725	0.011
0.735	0.724	0.011
0.733	0.723	0.010
0.732	0.722	0.010
0.730	0.721	0.009
0.728	0.720	0.009
0.727	0.718	0.009
0.725	0.717	0.008
0.723	0.716	0.008

Table 4.22 (Velocity network)

4.15 Conclusion

In this chapter, two networks, for depth and for velocity were developed as depicted by fig. 4.1 and specified mathematically by equations 4.1 and 4.2. Training was performed using the backpropagation algorithm and a bipolar sigmoid activation function. Using some generated ebb tide data, the simulations were first performed for the depths over a length of 600 Km in order to obtain information about a suitable architecture for the hidden layer. Results indicated a suitable architecture would be one with seven neurons in the hidden layer. Following this, further simulations were performed over 1200 Km (in order to avoid overfitting in the training) to obtain optimum values for the learning rate and momentum term which were 0.05 and 0.4 respectively. With the depth network fully specified, the

training simulations were performed again with an error (difference between the neural network simulation and the finite difference scheme generated data) of less than 0.003 m and a RMSE of 0.0014. It was noted that the major changes to the weights in the network as a result of training were limited mostly to those between the input and hidden layers. Validation of the depth network was again performed over 1200 Km using this time, some flood tide (unseen) generated data. The maximum difference between the generated data and the simulation was this time less than 0.008 m. The RMSE and ME were respectively 0.0065 and 0.0043 which compared favourably with that of Dibike [31] using a slightly similar model who obtained 0.0048 for the RMSE and 0.04 for the ME. The depth network is fully specified by fig.4.1 and the weights in table 4.9.

The process of determining the architecture for the velocity network followed the same procedure as that for the depth network. In the course of training, the maximum discrepancy between the neural network and the generated ebb tide data was less than 0.016 m/s and a RMSE of 0.0011. As for the depth training, it was observed that the changes to the input to hidden layer weights were much more dramatic than changes to the weights on the other connections. In fact, the changes between the input and hidden layers were about two to three times the changes between the other units. Validation of the velocity network was as for the depth, using flood tide data regenerated again over 1200 Km. An extremely good fit was obtained with a maximum error of 20 mm/s. The RMSE and ME values of 0.0079 and 0.0067 again compared favourably with the model of Dibike [31] who obtained 0.0085 and 0.05 for the RMSE and ME respectively.

Finally, the neural network simulations were compared with the numerical model of Johns [51]. Using the training phase results, the average and maximum differences in depth were 0.007 m and 0.013 m whilst for the velocity, they were 0.002 m/s. and 0.021 m/s. The corresponding results for the validation phase were 0.012 m and 0.033 m for depth and 0.008 m/s and 0.023 m/s for the velocity. The neural network simulation was therefore a very good 'fit' to the numerical model.

The velocity network is fully specified by the weights in table 4.18 and fig. 4.1.

Chapter 5

Neural network model with bottom friction

5.1 Introduction

The neural network model developed in the last chapter was somewhat idealized in that it ignored the effects of bottom friction (cf. Chapter 3). In this chapter, the investigations attempt to correct for that admission and in so doing, create a model that is much more representative of river flows. In fact, although the one dimensional shallow water equations (St. Venant Equations) are widely used to represent river flows and indeed have been simulated using neural networks, these simulations have not included the effects of bottom friction. In the work discussed here on the models with bottom friction, the methodology and pseudo code of the previous chapter are also applicable to this chapter. Consequently, they will not be reproduced at this juncture. Additionally, the same values for the learning rate, momentum term and number of neurons in the hidden layer were reused in this analysis. Further, the same computer programs as discussed in the previous chapter are also reused for the models with a bottom friction, the latter having a value of 0.0026 unless otherwise indicated. In this chapter the work concentrates on the development of a neural network model incorporating the effects of bottom friction and the subsequent training and validation of the model. However, due to a rather inadequate performance by the network, a re-examination of the parameters and architecture is required followed by re-training and re-validation. Having established a network with suitable performance, this network is then tested further with a variation of the (non-architectural) parameters. The chapter concludes with the application of the network to the confluence of the rivers Thames and Medway.

5.2 Training and validating the neural network with bottom friction

5.2.1 Training the depth network

After running a simulation for 12000 iterations in an ebb tide regime, that is an estuarine velocity of 0.5 m/s, the following plots of the depth (target and ANN estimate) were obtained:

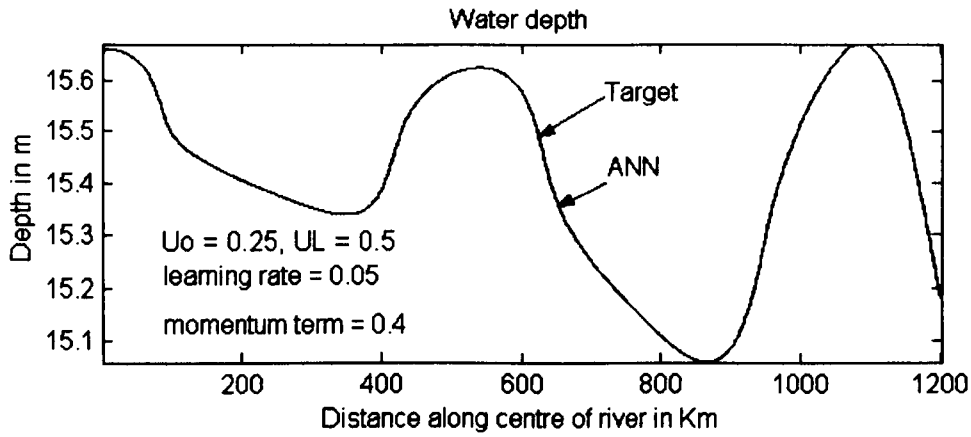

Figure 5.1: Ebb tide simulation of the depth

with depths at various points along the river being:

	0 Km	349 Km	542 Km	866 Km	1086 Km	1199 Km
ANN	15.6483	15.3390	15.6262	15.0583	15.6721	15.1829
Target	15.6624	15.3390	15.6262	15.0585	15.6721	15.1827
Difference	-0.0141	0.0000	0.0000	-0.0002	0.0000	0.0002

Table 5.1: Ebb tide depth variations

and a table depicting the errors in training:

Errors from training			
RMSE	ME	ESD	RPE
0.00083299	0.00026694	0.000025282	0.0017

Table 5.2: Ebb tide depth training errors

Clearly from fig. 5.1 and tables 5.1 and 5.2, the ANN is appearing to perform well during the training phase of the depths with error ratios of:

$$\frac{ME}{RMSE} \sim \pm 0.321, \quad \frac{ESD}{RMSE} \sim \pm 0.030 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 2.041$$

5.2.2 Training the velocity network

The simulation was run again for 12000 iterations in an ebb tide regime (with the same hydrodynamical data) resulting in the following plot of the velocities of the target and ANN estimate:

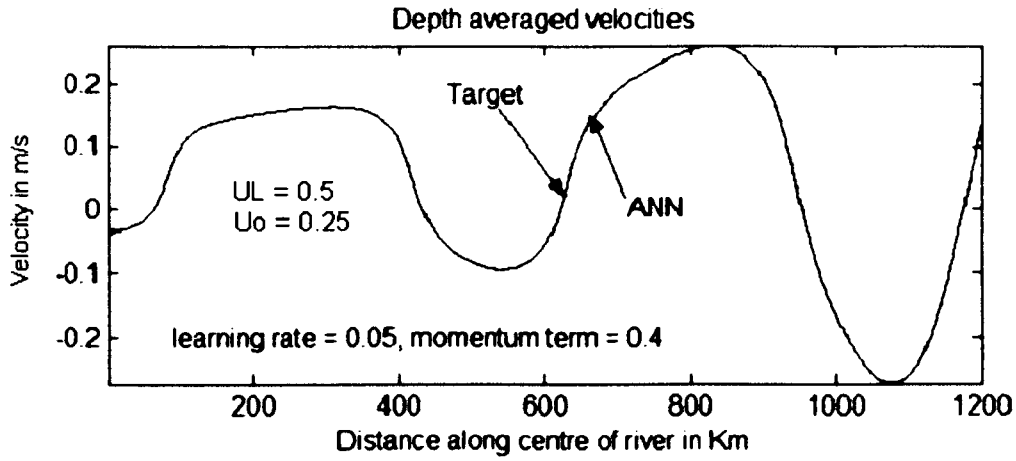

Figure 5.2: Ebb tide simulation of the velocity

a table containing velocities at various locations:

	0 Km	310 Km	542 Km	835 Km	1078 Km	1200 Km
ANN	-0.0279	0.1644	-0.0950	0.2620	-0.2737	0.1338
Target	-0.0356	0.1644	-0.0950	0.2621	-0.2736	0.1354
Difference	0.0077	0.0000	0.0000	-0.0001	-0.0001	-0.0016

Table 5.3: Ebb tide velocity variations

and a table of errors:

Errors from training			
RMSE	ME	ESD	RPE
0.0004944	0.00021267	0.000015547	0.0014

Table 5.4: Ebb tide velocity training errors

The error ratios were:

$$\frac{ME}{RMSE} \sim \pm 0.43, \quad \frac{ESD}{RMSE} \sim \pm 0.031 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 2.832$$

Although different quantities (that is, depth and velocity) are being compared, it is interesting to note, from tables 5.2 and 5.4, that the velocity RMSE and velocity ESD are both about 60% of their depth counterparts whilst for the ME and RPE, they are approximately 80% of the depth equivalents. This is encouraging since (cf. section 4.1) any errors occurring in the first (depth) network are propagated into the second (velocity) network. However, as previously noted, the effects of bed friction may well be helping to curtail any such error propagation.

Again, as for the depths, from fig. 5.2 and tables 5.3 and 5.4 it can be observed that the neural network is performing well on this second stage of the training phase. It now remained to test the networks on unseen data to verify that the model of Chapter 4, wherein bed friction was absent, to see if it could generalize to this more realistic scenario.

5.2.3 Validation of the depth network

As for the validation of the models in Chapter 4, a flood tide regime with an estuarine velocity of -0.5 m/s. was adopted. Over a validation period of 12000 iterations, the simulation produced the following results:

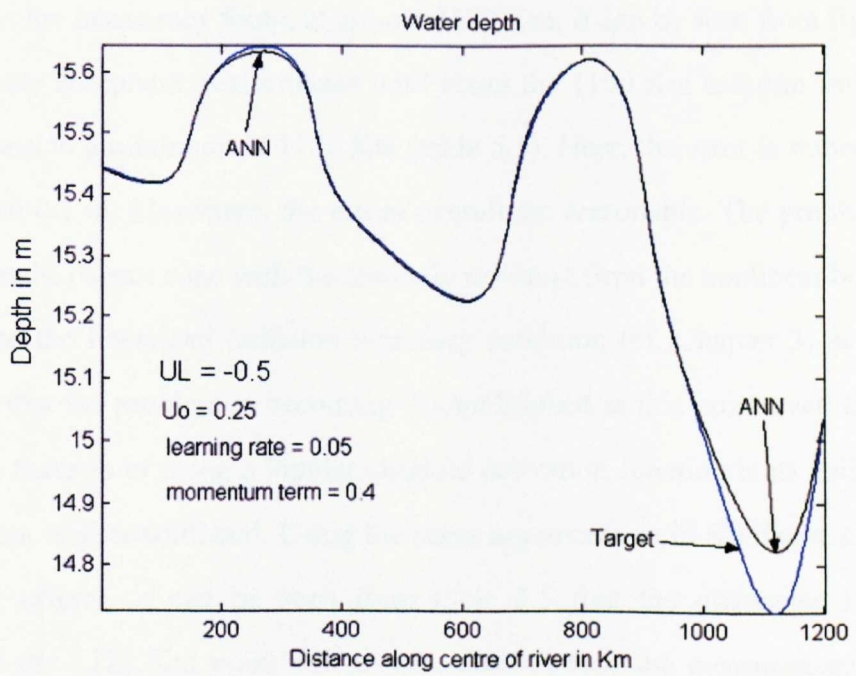

Figure 5.3: Flood tide simulation of the depth

	0 Km	88 Km	270 Km	609 Km	814 Km	1122 Km	1199 Km
ANN	15.4414	15.4171	15.6339	15.2258	15.6214	14.8235	15.0312
Target	15.4446	15.4197	15.6428	15.2284	15.6222	14.7239	15.0325
Diff.	-0.0032	-0.0026	-0.0089	-0.0026	-0.0008	0.0996	-0.0013

Table 5.5: Flood tide depth variations

Errors from validation			
RMSE	ME	ESD	RPE
0.0457	0.0209	0.0013	0.1394

Table 5.6: Flood tide validation depth errors

Although the error ratios of:

$$\frac{ME}{RMSE} \sim \pm 0.457, \quad \frac{ESD}{RMSE} \sim \pm 0.028 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 3.050$$

are of the correct order of magnitude (cf. Chapter 4), it is misleading as all of the errors have been magnified from the training phase to the validation phase. The depth training RMSE, ME, ESD and RPE have been magnified respectively by about 55, 78, 51 and 82. This can only be due to the inaccuracy found at around 1122 Km. It can be seen from fig. 5.3 that the ANN has a very acceptable performance until about the 1100 Km location from whence its accuracy decays to a minimum at 1122 Km (table 5.5). Here, the error is rather large, being of the order of 0.1 m. Elsewhere, the errors overall are reasonable. The problem is that the ANN as it stands, cannot cope with the anomaly resulting from the nonlinear bottom friction effects near to the linearized radiation boundary condition (cf. Chapter 3) at 1199 Km. It may well be that the problem is becoming ill-conditioned at this point even though one of the attractive features of using a bipolar sigmoid activation function is its ability to help to keep a problem well conditioned. Using the same arguments as in Section 4.8.2 (Admiralty measurement errors), it can be seen from table 5.5 that the difference is only really significant at the 1122 Km point where it is almost twice the measurement error of the Admiralty.

5.2.4 Validation of the velocity network

To check that the anomaly was not just peculiar to the depth network, a similar simulation was run for the velocities, which produced the following results:

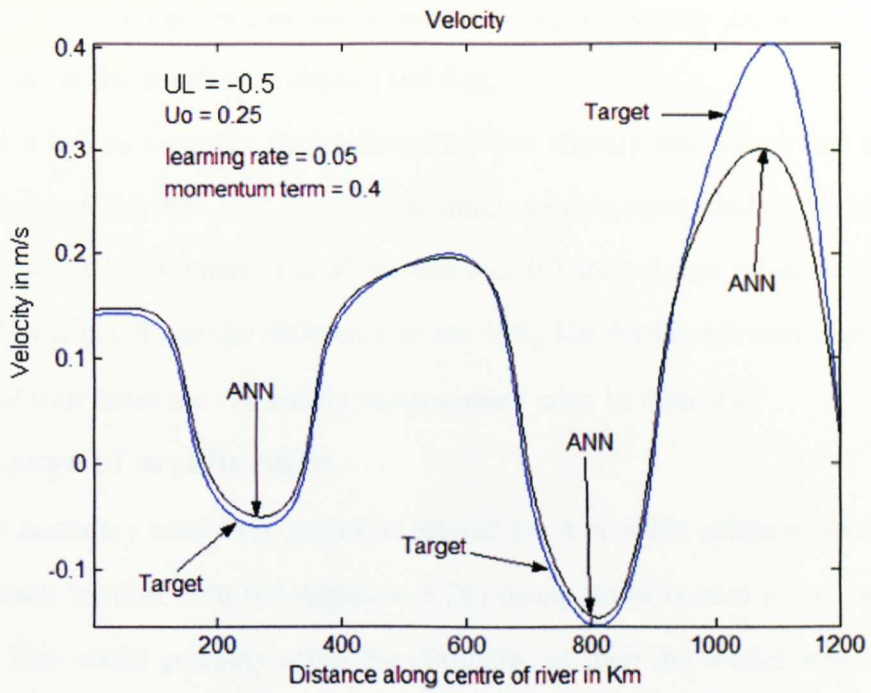

Figure 5.4: Flood tide simulation of the velocity

	0 Km	45 Km	270 Km	573 Km	813 Km	1082 Km	1200 Km
ANN	0.1458	0.1478	-0.0506	0.1959	-0.1456	0.3014	0.0277
Target	0.1402	0.1417	-0.0597	0.1997	-0.1535	0.4034	0.0382
Diff.	0.0056	0.0061	0.0091	-0.0038	0.0079	-0.1020	-0.0105

Table 5.7: Flood tide velocity variations

Errors from validation			
RMSE	ME	ESD	RPE
0.0128	0.0099	0.00042433	0.0656

Table 5.8: Flood tide velocity validation errors

As for the depth analysis, although the error ratios of:

$$\frac{ME}{RMSE} \sim \pm 0.773, \quad \frac{ESD}{RMSE} \sim \pm 0.033 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.125$$

are within the range of expected values, again all of the errors have been increased from the training phase to the validation phase. The velocity training errors RMSE, ME, ESD and RPE have been respectively magnified by factors of approximately 25, 47, 27 and 47. This can be attributed to the anomaly at about 1100 Km.

From fig. 5.4 it can be seen that the performance was slightly worse than that for the depth up until approximately 900 Km with the anomaly getting even slightly greater [than the depth one] at about 1100 where it is of the order of 0.1 m/s. Again using the arguments of Section 4.8.2, it is noted that the difference at the 1082 Km location is very significant being of the order of four times the Admiralty measurement error in velocity.

5.3 A change of architecture

The radiation boundary condition cannot be altered but a possible solution would have been to let the bottom friction term (cf. equation 3.26) decay towards zero as the boundary was approached. This could possibly solve the difficulty as then the model would degenerate into the one [zero bottom friction] model, near to the boundaries, simulated successfully in the previous chapter.

5.3.1 A change to the hidden layer

However, rather than follow this line of enquiry, it was preferred that a further examination of the architecture of the ANN for this friction model as well as other parameters such as the learning rate and momentum term would be carried out. Simulating with different learning rates and momentum terms it appeared that the choice of 0.05 and 0.4 for these two parameters using the method suggested by Haykin (cf. Chapter 4) was right. It remained therefore to examine the architecture of this friction model again to see if changing the number of neurons in the hidden layer would help. This in fact seems a reasonable assumption because figs. 5.3 and 5.4 suggest that the networks (as they stand) lack the flexibility to generalize on this more complex (bottom friction) model than the one in the previous chapter.

The model was simulated using three, five, seven, nine, eleven and thirteen (not shown) neurons and a plot of their RMSE curves over 2000 iterations (fig. 5.5) was obtained.

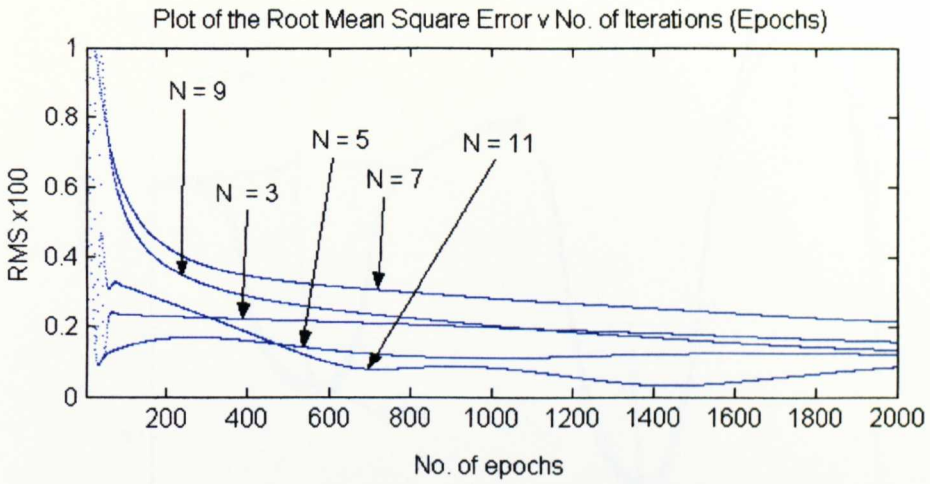

Figure 5.5: RMSE plots of different architectures

It is apparent that the curves representing seven and nine neurons are the best. The depth and velocities were also re-simulated using these different architectures and apart from that with nine neurons, all of these different architectures aggravated the problem, in particular, for an architecture of eleven neurons, the depth anomaly was almost doubled. The plots of the simulations (validation) for the depth and velocity using seven and nine neuron architectures follow:

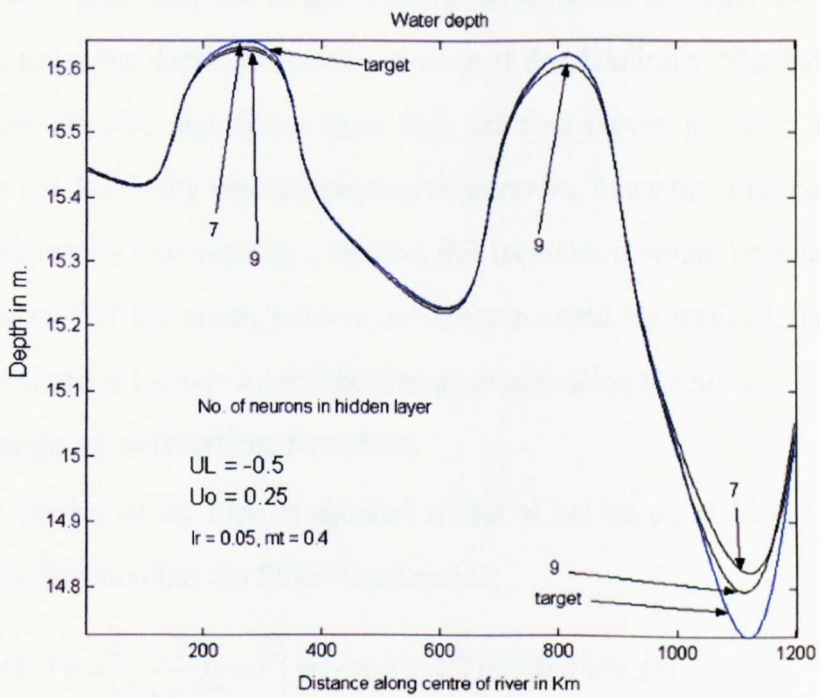

Figure 5.6: Flood tide depth simulations for different architectures

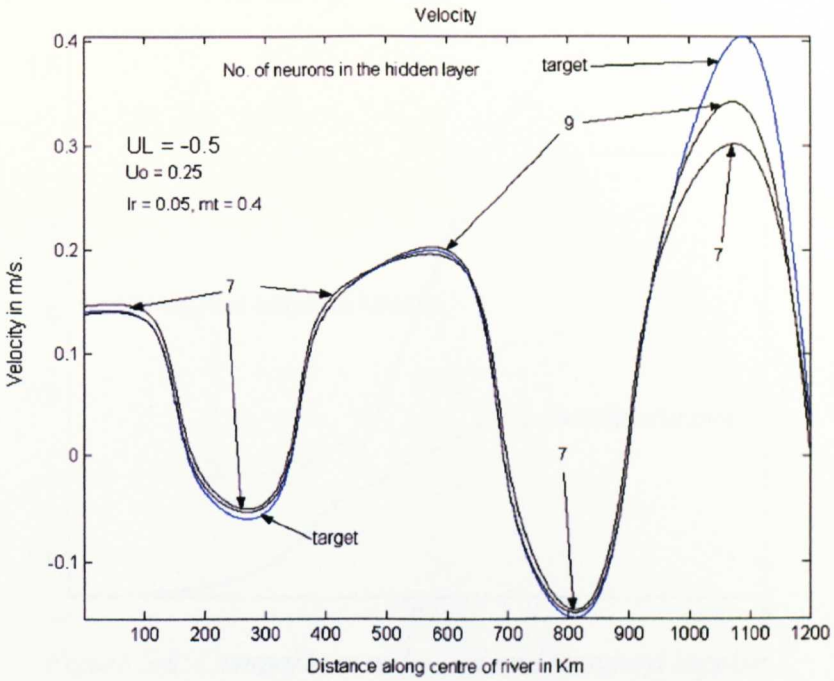

Figure 5.7: Flood tide velocity simulations for different architectures

It can be seen from figs. 5.6 and 5.7 that an architecture with seven neurons is marginally better throughout the domain of interest until the anomaly at about 1100 Km is approached, at which point the nine neuron model reduces the discrepancy by about 50%. Once again using the arguments of Section 4.8.2, it is noted that depth differences are significant at the 1100 Km location. Here they are of the order of twice (seven neurons) and one and a half (nine neurons) times the depth measurement error of the Admiralty. Similarly, the velocity differences there are also significant since they are four (seven neurons) and three (nine neurons) times the Admiralty velocity measurement errors. Since the difference between the two architectures really only manifests itself at this location, it would be consistent with the model in Chapter 4 if the seven neuron architecture could be retained. To this end, the enquiry was widened to include a possible change of activation function.

5.3.2 A change of activation function

An alternative version of the bipolar sigmoid is that of LeCun et. al. [65], LeCun [63] and LeCun [64]. The function has the following formula:

$$a\tanh(bx) \equiv a\left(\frac{2}{1+e^{-2bx}}-1\right) \text{ where a = 1.7159 and b = 2/3} \qquad \textbf{(5.1)}$$

A comparative plot of the two sigmoids produces:

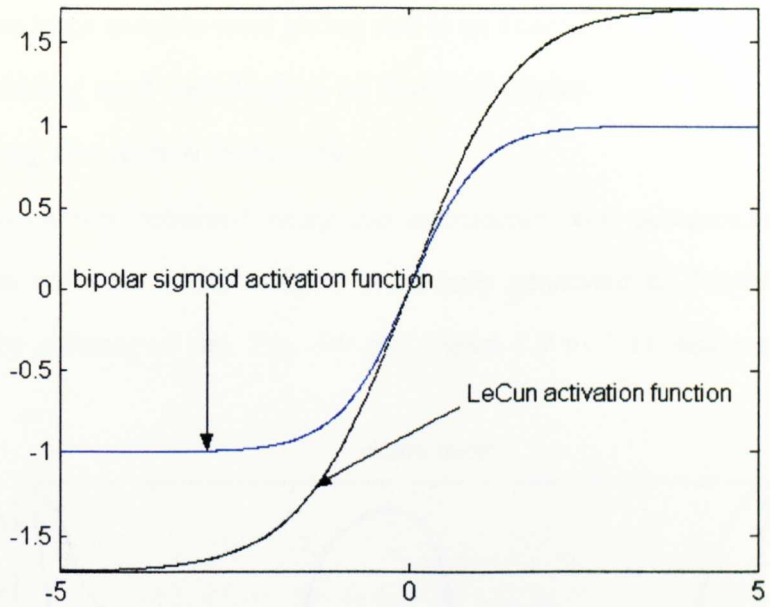

Figure 5.8: Comparison of LeCun and standard bipolar

The coefficient 'a' in equation **(5.1)** determines the slope of the activation function. Increasing the value of 'a' increases the slope and vice versa. A high value for 'a' results in a form similar to the step function whilst a low value tends to retard the convergence rate. A value of 1.5 aids rapid convergence. A further attractive feature of this activation function (also true for other bipolars) is that its derivative does not significantly increase the computational workload. The Matlab code was amended to incorporate this change of activation function and subsequently tested. The LeCun activation function was incorporated into the code for the forward pass but the backward pass part of the algorithm remained using the standard bipolar activation function code. Unfortunately, it produced plots of the neural network estimates of the depth that were highly oscillatory resembling a classic tan curve 'cropped' at the top and bottom. This immediately indicated that some of the weighted values from the output unit were exceeding the upper and lower limits of the function. Indeed, upon writing a small piece of test code using the weights being generated by the original software, it was observed that the output was similar in shape to a standard Heaveside type function with upper and lower limits of 1.7159 and – 1.7159 respectively. Although the initial weights generated by Matlab were random and normally distributed, they were probably, initially too large. After some experimentation, it was found that

dividing all of these initial weights by a factor of ten resolved the problem. This was probably because large weights were giving rise to an excessive variance in the output.

5.4 Re-training and validation of the networks

5.4.1 Training the depth network

The depth network was retrained using the architecture and parameters of the previous section but now with the initial weights (randomly generated by Matlab), to the hidden layer, divided by a factor of ten. Fig. 5.9 and tables 5.9 to 5.11 depict the results of this training:

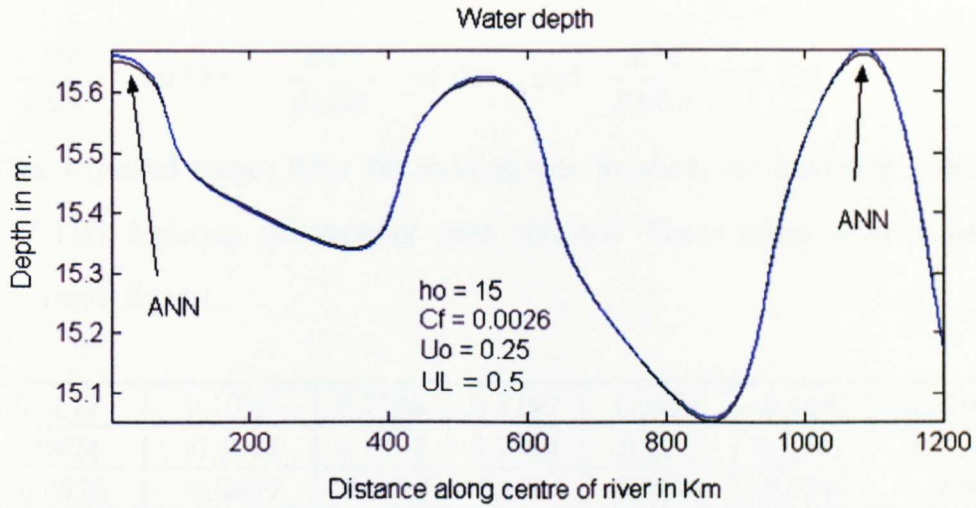

Figure 5.9: Re-trained ebb tide depth simulation

The network appears to be able to simulate the target depth very well with just small deviations at the locations as shown in table 5.9 below. All of these deviations appear to be less than 0.01 m and tend to be an underestimate, except at 350 Km, where the network is overestimating.

	0 Km	350 Km	543 Km	867 Km	1086 Km	1200 Km
ANN	15.6534	15.3427	15.6210	15.0524	15.6636	15.1753
Target	15.6624	15.3390	15.6262	15.0585	15.6721	15.1753
Diff.	-0.0090	0.0037	-0.0052	-0.0061	-0.0085	0.0000

Table 5.9: Ebb tide depth variations

It is noted now that none of the differences are significant in comparison to the Admiralty measurement errors (see Section 4.8.2).

Errors from training			
RMSE	ME	ESD	RPE
0.0042	0.0037	0.00017	0.0241

Table 5.10: Ebb tide depth re-training errors

The error ratios:

$$\frac{ME}{RMSE} \sim \pm 0.881, \quad \frac{ESD}{RMSE} \sim \pm 0.041 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.738$$

were in the expected range. After the training was finished, the following results (tables 5.11a to 5.11f) depicting the weights were obtained. These tables need to be read in conjunction with fig. 4.1:

0.0617	0.1059	0.2186	0.4297	0.3041	-0.1442	-0.2701
0.0474	-0.0044	0.1414	0.3984	-0.1224	0.0593	-0.0936
-0.0825	-0.0669	-0.3487	-0.1564	-0.1789	-0.0341	-0.0955

Table 5.11a Final weights from input layer to hidden layer

-0.0637	-0.1054	0.1373	0.1634	0.0672	0.0269	0.1536
-0.1003	-0.0072	0.0180	0.0825	-0.0508	0.0625	0.0434
-0.0186	0.0279	-0.0542	0.0231	0.0856	-0.1047	-0.1917

Table 5.11b Original weights from input layer to hidden layer

0.1254	0.2113	0.0813	0.2663	0.2369	-0.1711	-0.4236
0.1477	0.0028	0.1234	0.3159	-0.0716	-0.0032	-0.1370
-0.0640	-0.0948	-0.2945	-0.1795	-0.2646	0.0707	0.0962

Table 5.11c Change from original

Final weight matrix from Bias unit(s) to hidden layer:		Original weight matrix from Bias unit(s) to hidden layer:		Change from original:
0.1452		0.1789		-0.0337
0.0429		0.0391		0.0039
-0.0961		0.0020		-0.0981
-0.0826		-0.0406		-0.0420
-0.1761		-0.1535		-0.0226
0.0158		0.0221		-0.0063
-0.1418		-0.1374		-0.0043

Table 5.11d: Matrix of weights for Bias unit to hidden layer

Final weight matrix from hidden layer to output layer:		Original weight matrix from hidden layer to output layer:		Change from original:
0.4258		0.0470		0.3788
0.2121		0.1274		0.0846
0.7122		0.0639		0.6484
0.8188		0.1381		0.6807
0.3569		0.1320		0.2249
-0.1294		-0.0909		-0.0385
-0.4629		-0.2306		-0.2323

Table 5.11e: Matrix of weights from hidden layer to output layer

Final weight matrix from Bias unit(s) to output layer:		Original weight matrix from Bias unit(s) to output layer:		Change from original:
0.0660		-0.0839		0.1499

Table 5.11f: Matrix of weights from Bias unit to output layer

Inspection of tables 5.11a to 5.11c indicate that nine weights have changed sign with the largest change in value being -0.4236. Comparing the final weights to the original, the original weights have been 'magnified' overall by a factor of approximately 2.861, the largest factor being 6.771 and the smallest, 0.3257. Continuing, comparing the original and final weight values for the bias unit to the hidden layer, it is noted that only one weight changed sign, in fact the one whose value was magnified the most. The original value of 0.0020 changed sign and was magnified by a factor of about 48. The smallest weight

magnification was 0.715 but on average, the original weights were magnified by a factor of around 7.841. A comparison of the original and final weights from the hidden layer to the output layer indicate that the changes were more modest with not a single change of sign. The smallest weight magnification was around 1.424, the largest, 11.146 and overall, 4.848. In contrast, the change to the weight from the bias unit to the output layer was minimal with just a change of sign and a magnification factor of 0.787.

5.4.2 Training the velocity network

The velocity network was retrained as per the depth one using the same parameters. Fig. 5.10 and tables 5.12 to 5.13 depict the results of this training:

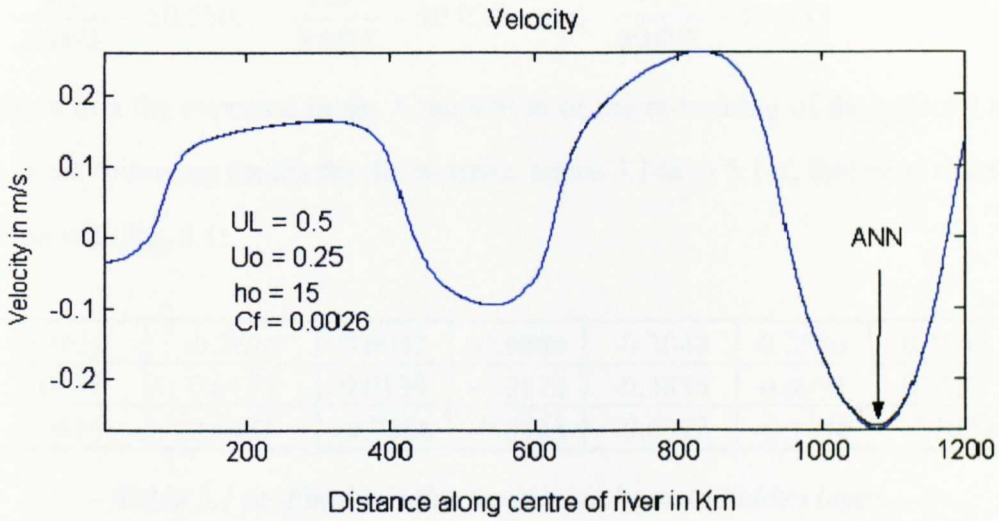

Figure 5.10: Re-trained ebb tide velocity simulation

The network appears to be able to simulate the target velocity very well with just small deviations at the locations as shown in table 5.12 below. All the deviations are less than 0.01 m and tend to be a mixture of over and under estimations. The greatest deviation occurs at 1079 Km.

	0 Km	311 Km	544 Km	835 Km	1079 Km	1200 Km
ANN	-0.0355	0.1650	-0.0946	0.2617	-0.2689	0.1354
Target	-0.0356	0.1644	-0.0950	0.2620	-0.2736	0.1354
Diff.	0.0001	0.0006	0.0004	-0.0003	0.0047	0.0001

Table 5.12: Ebb tide velocity variations

Again it is noted that none of the differences are significant in comparison to the Admiralty measurement errors (see Section 4.8.2).

Errors from training			
RMSE	ME	ESD	RPE
0.0012	0.00066	0.00003	0.0044

Table 5.13: Ebb tide velocity re-training errors

The usual error ratios are then:

$$\frac{ME}{RMSE} \sim \pm 0.550, \quad \frac{ESD}{RMSE} \sim \pm 0.025 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 3.667$$

which are within the expected range. Completion of the re-training of the velocity network resulted in the following results for the weights, tables 5.14a to 5.14f, that need to be read in conjunction with fig. 4.1:

-0.1619	-0.3614	0.6041	-0.0880	-0.3048	0.3586	0.1948
0.0098	0.0171	0.0139	-0.0174	-0.3835	0.0995	0.0971
0.0615	0.0487	-0.2536	-0.0745	-0.0372	-0.1048	-0.1039

Table 5.14a Final weights from input layer to hidden layer

-0.0209	-0.1345	0.1870	-0.0767	-0.0964	0.0257	-0.0117
0.0756	0.1482	-0.1209	-0.0107	-0.2379	-0.0184	0.0168
0.0376	0.0033	-0.0783	-0.0977	-0.0838	-0.0168	-0.0501

Table 5.14b Original weights from input layer to hidden layer

-0.1410	-0.2268	0.4171	-0.0113	-0.2084	0.3329	0.2065
-0.0658	-0.1311	0.1348	-0.0066	-0.1456	0.1179	0.0802
0.0240	0.0455	-0.1753	0.0232	0.0467	-0.0880	-0.0537

Table 5.14c Change from original

Final weight matrix from Bias unit(s) to hidden layer:		Original weight matrix from Bias unit(s) to hidden layer:		Change from original:
-0.0839		-0.0890		0.0051
0.0076		0.0139		-0.0064
-0.0310		-0.0236		-0.0074
0.0023		-0.0075		0.0099
-0.0074		-0.0359		0.0284
-0.1856		-0.2078		0.0221
-0.0135		-0.0144		0.0008

Table 5.14d: Matrix of weights for Bias unit to hidden layer

Final weight matrix from hidden layer to output layer:		Original weight matrix from hidden layer to output layer:		Change from original:
-0.1502		-0.0705		-0.0797
-0.2962		0.0508		-0.3470
0.5653		-0.0421		0.6074
-0.0535		0.0229		-0.0764
-0.3987		-0.0959		-0.3028
0.3630		-0.0146		0.3776
0.2346		0.0745		0.1601

Table 5.14e: Matrix of weights from hidden layer to output layer

Final weight matrix from Bias unit(s) to output layer:		Original weight matrix from Bias unit(s) to output layer:		Change from original:
0.0836		0.1393		-0.0557

Table 5.14f Matrix of weights from Bias unit to output layer

Analysing tables 5.14a to 5.14c, containing the weights from the input to hidden layers, the largest magnification factor was 16.65, the smallest, 0.115 and the average, 4.405. The network training changed the signs of three of the weights. There were no dramatic changes in size of the weights, the largest change being 0.4171 and the smallest being 0.0066 (ignoring sign). For the changes to the bias to hidden weights (table 5.14d), changes were much more moderate with just one sign change. The average magnification factor, a reduction in this case, from the bias to the hidden layers, was 0.735. The changes to the

weights between the hidden and output layers were much more dramatic, involving four sign changes. The smallest magnification was 2.130 and the largest, 24.863, with an average magnification of 7.985.

As for the bias unit to the hidden layer, the change was quite modest involving a reduction in size using a factor of 0.600

5.4.3 Validating the depth network

As usual a flood tide regime was used for the validation with an estuarine velocity of -0.5 m/s. simulated over a period of 12000 iterations which resulted in the data as depicted in fig. 5.11 and tables 5.15 and 5.16.

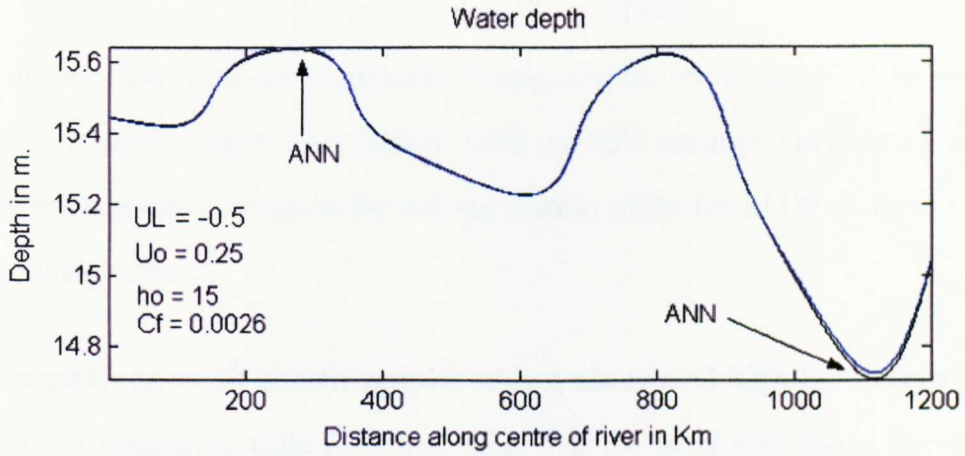

Figure 5.11: Re-trained flood tide depth simulation

Once more it is noted that none of the differences are significant in comparison to the Admiralty measurement errors (see Section 4.8.2).

	0 Km	89 Km	270 Km	609 Km	815 Km	1119 Km	1200 Km
ANN	15.4450	15.4231	15.6357	15.2296	15.6179	14.7054	15.0393
Target	15.4450	15.4197	15.6427	15.2284	15.6222	14.7233	15.0393
Diff.	0.0000	0.0034	-0.0070	0.0012	-0.0043	-0.0179	0.0000

Table 5.15: Flood tide depth variations

As can be seen from fig. 5.11, the neural network is performing well on the unseen data. The maximum difference between the target value and the ANN value appears to be around the usual point, 1119 Km, but is of a very reasonable sized 0.0179 m. It is about twice the

size of the maximum difference in the training phase. In fact, generally all of the differences are of the order of twice those in the training phase.

Errors from training			
RMSE	ME	ESD	RPE
0.0071	0.0052	0.00024	0.0345

Table 5.16: Flood tide depth re-training errors

The error ratios, from table 5.16, are:

$$\frac{ME}{RMSE} \sim \pm 0.732, \quad \frac{ESD}{RMSE} \sim \pm 0.034 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.859$$

which are again of the expected magnitudes. Comparing the errors in table 5.16 with those from the training session, table 5.10, the ME, ESD and RPE are approximately 1.4 times the size of their corresponding values in the training session whilst the RMSE is about 1.7 times the corresponding value.

5.4.4 Comparison with depth results using standard bipolar activation

Interestingly, on comparing table 5.15 with table 5.5, the latter containing the validation results using the standard bipolar activation function, it is seen that the maximum difference (the anomaly) of 0.0996 at 1122 Km has been reduced by about a factor of 5.5. Further, comparing table 5.16 with table 5.6, the RMSE, ME, ESD and RPE have been reduced by factors of 0.155, 0.249, 0.185 and 0.247 respectively.

5.4.5 Finalized weights of the depth network

After validating the network, the following table of weights, that needs to be read in conjunction with fig. 4.1, was obtained:

Final weight matrix from input layer to hidden layer:							
	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
X_0	0.1452	0.0429	-0.0961	-0.0826	-0.1761	0.0158	-0.1418
X_1	0.0617	0.1059	0.2186	0.4297	0.3041	-0.1442	-0.2701
X_2	0.0474	-0.0044	0.1414	0.3984	-0.1224	0.0593	-0.0936
X_3	-0.0825	-0.0669	-0.3487	-0.1564	-0.1789	-0.0341	-0.0955

Final weight matrix from hidden layer to output layer:								
	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
Z	0.0660	0.4258	0.2121	0.7122	0.8188	0.3569	-0.1294	-0.4629

Table 5.17: Finalized weights of the re-trained depth network

5.4.6 Validating the velocity network

Using the same parameters as for the depth validation, the velocity was validated over a period of 12000 iterations with resultant data as in fig. 5.12 and tables 5.18 and 5.19.

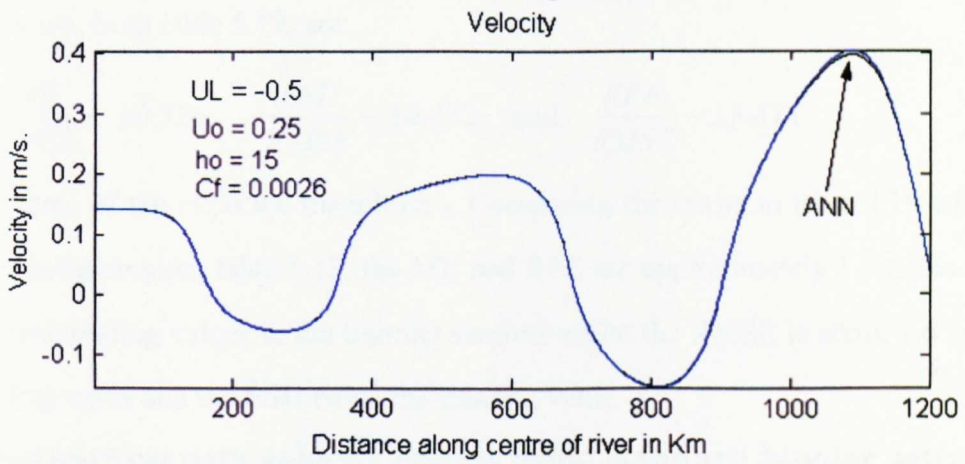

Figure 5.12: Re-trained flood tide velocity simulation

Again it is noted that none of the differences are significant in comparison to the Admiralty measurement errors (see Section 4.8.2).

	0 Km	43 Km	270 Km	574 Km	811 Km	1089 Km	1200 Km
ANN	0.1407	0.1423	-0.0595	0.2002	-0.1525	0.3975	0.0370
Target	0.1402	0.1417	-0.0597	0.1997	-0.1535	0.4044	0.0382
Diff.	0.0005	0.0006	0.0002	0.0005	0.0010	-0.0069	-0.0012

Table 5.18: Flood tide velocity variations

Fig. 5.12 demonstrates that the neural network has performed well on the unseen validation data, the maximum difference between the target value and the ANN value being less than 0.007 m. This occurs at approximately the same position of the anomaly in the velocity validation using the standard bipolar function (fig. 5.4). This maximum deviation of 0.0069 is about 1.5 times the maximum deviation in the training phase (cf. table 5.12). In fact, most of the deviations in table 5.18 are more or less 1.5 times their equivalents from the training phase data in table 5.12.

Errors from training			
RMSE	ME	ESD	RPE
0.0019	0.0010	0.00006	0.0066

Table 5.19: Flood tide velocity re-training errors

The error ratios, from table 5.19, are:

$$\frac{ME}{RMSE} \sim \pm 0.526, \quad \frac{ESD}{RMSE} \sim \pm 0.032 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 3.474$$

which are again of the expected magnitudes. Comparing the errors in table 5.19 with those from the training session, table 5.13, the ME and RPE are approximately 1.5 times the size of their corresponding values in the training session whilst the RMSE is about 1.6 times the corresponding value and the ESD twice the training value.

5.4.7 Comparison with velocity results using standard bipolar activation

Comparing table 5.18 with table 5.7, the latter containing the validation results using the standard bipolar activation function, it can be seen that the maximum difference (the anomaly) of 0.1020 at 1082 Km has been reduced by about a factor of 15. In fact, the differences have been reduced by factors between 8 and 15. Further, comparing table 5.19 with table 5.8, the RMSE, ME, ESD and RPE have been reduced by factors of 0.15, 0.1, 0.15 and 0.1 respectively.

5.4.8 Finalized weights of the velocity network

After validating the network, the following table of finalized weights was obtained:

Final weight matrix from input layer to hidden layer:							
	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
X_0	-0.0839	0.0076	-0.0310	0.0023	-0.0074	-0.1856	-0.0135
X_1	-0.1619	-0.3614	0.6041	-0.0880	-0.3048	0.3586	0.1948
X_2	0.0098	0.0171	0.0139	-0.0174	-0.3835	0.0995	0.0971
X_3	0.0615	0.0487	-0.2536	-0.0745	-0.0372	-0.1048	-0.1039

Final weight matrix from hidden layer to output layer:								
	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
Z	0.0836	-0.1502	-0.2962	0.5653	-0.0535	-0.3987	0.3630	0.2346

Table 5.20: Finalized weights of the re-trained velocity network

5.5 Testing the networks with different parameters

So far a neural network has been developed, as specified by fig. 4.1 and tables 5.17 and 5.20, that performs well for a particular still water depth, a fixed upstream (forcing) velocity and two different instances of the downstream velocity, namely, the tidal and the ebb flow velocities. It would be useful to test the system at this point, using different parametric values for the depth, friction and forcing velocities to analyse its 'universal approximation' capabilities. Strictly speaking, changing the still water depth or coefficient of bottom friction should require a complete retraining of the neural network. Five groups of tests were considered involving changes to the still water depth, coefficient of bottom friction, upstream velocities and downstream velocities as well as combinations of the two velocities.

5.5.1 Variation of the still water depth

Simulations were conducted with the still water depth changed first to 10 m and then 20 m for both flood tide and ebb tide regimes. The error measure results are depicted in table 5.21 at the end of this section.

Ebb tide: 10 m depth

With an upstream (forcing) velocity U_0 of 0.25 m/s, a downstream (estuarine) velocity U_L of 0.5 m/s and the friction coefficient set to 0.0026, as in the training phase, the still water

depth was changed to 10 m. Fig. 5.13, depicts the result of the simulations for the depth and velocity.

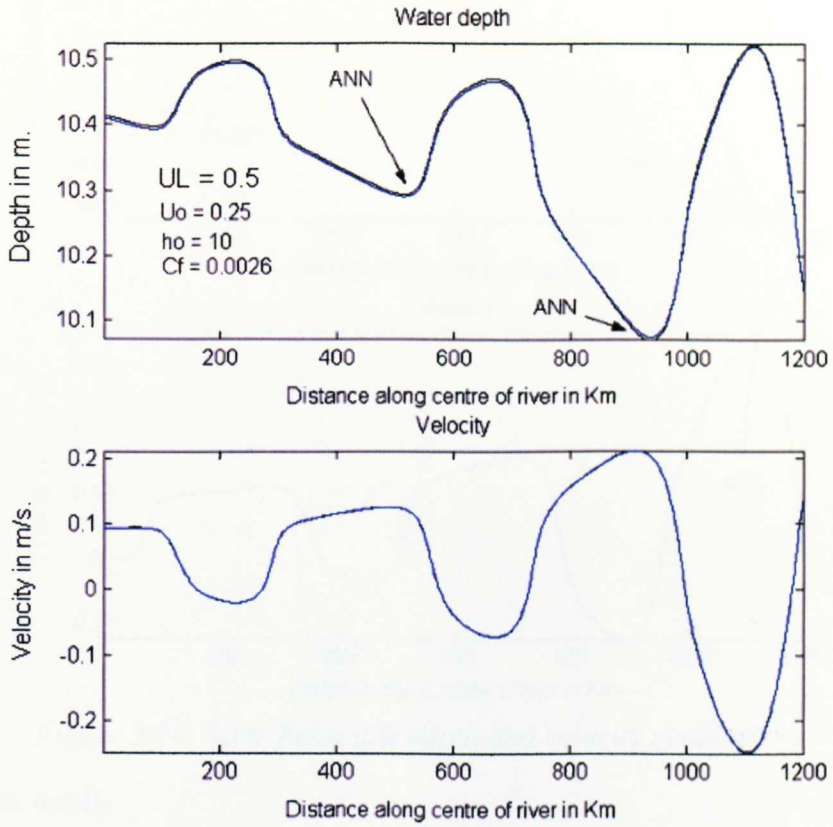

Figure 5.13: 10 m ebb tide depth and velocity simulations

There appears to be an extremely good agreement between the neural network simulation and the finite difference scheme plot, the velocity simulation being more accurate than that of the depth.

Flood tide: 10 m depth

With the parameters exactly as in the previous case but now with the downstream velocity U_L changed to one of a flood tide, in this case, -0.5 m/s, the simulation was run again. As can be seen from fig. 5.14 the velocity agreement is again very good but the depth is not quite as accurate as in the previous case between 1000 Km and 1200 Km.

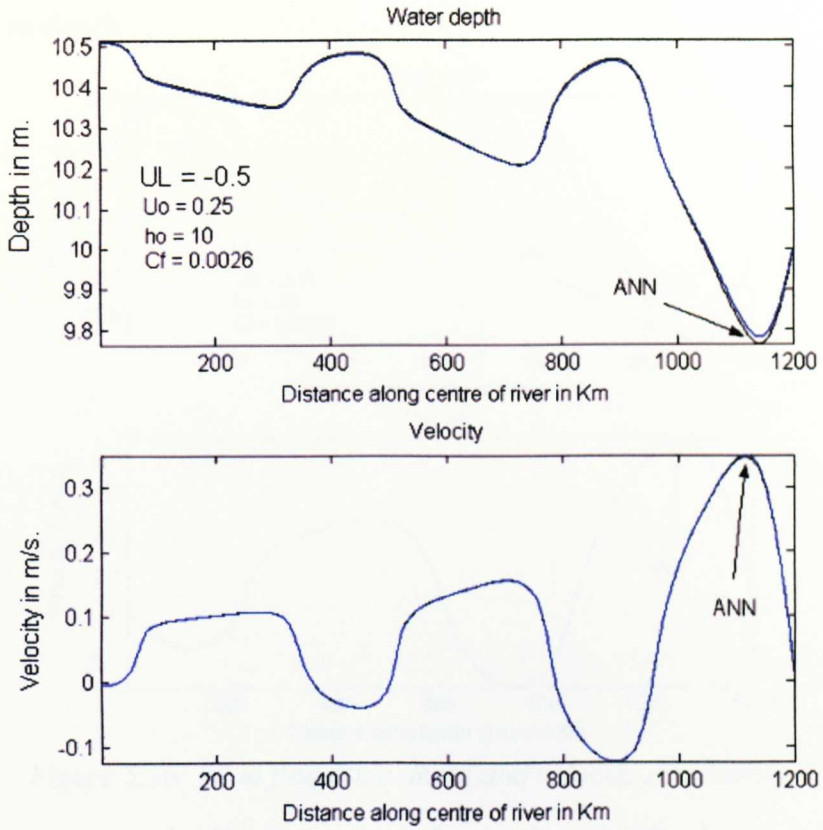

Figure 5.14: 10 m flood tide depth and velocity simulations

Ebb tide: 20 m depth

These 'test' simulations follow exactly the same procedure as for the 10 m case.

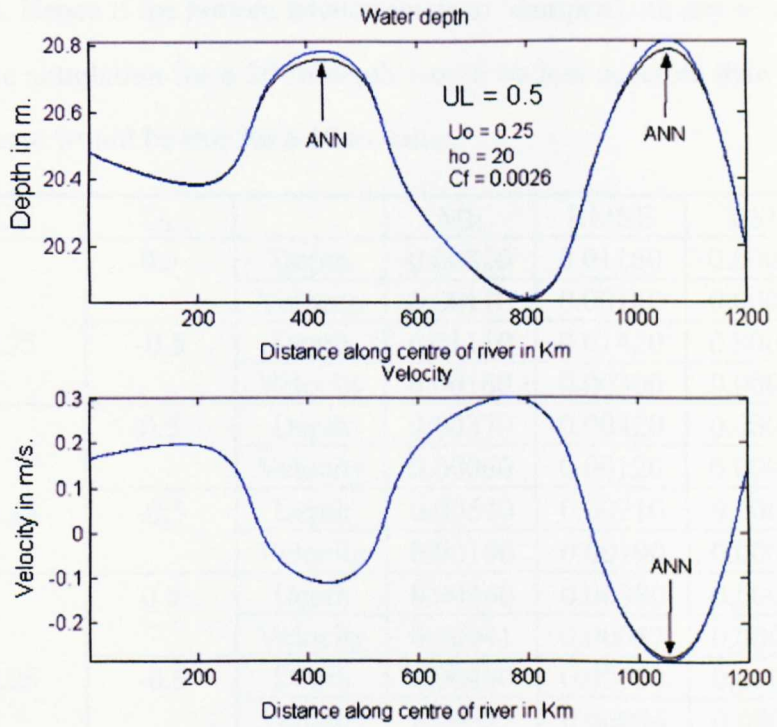

Figure 5.15: 20 m ebb tide depth and velocity simulations

Flood tide: 20 m depth

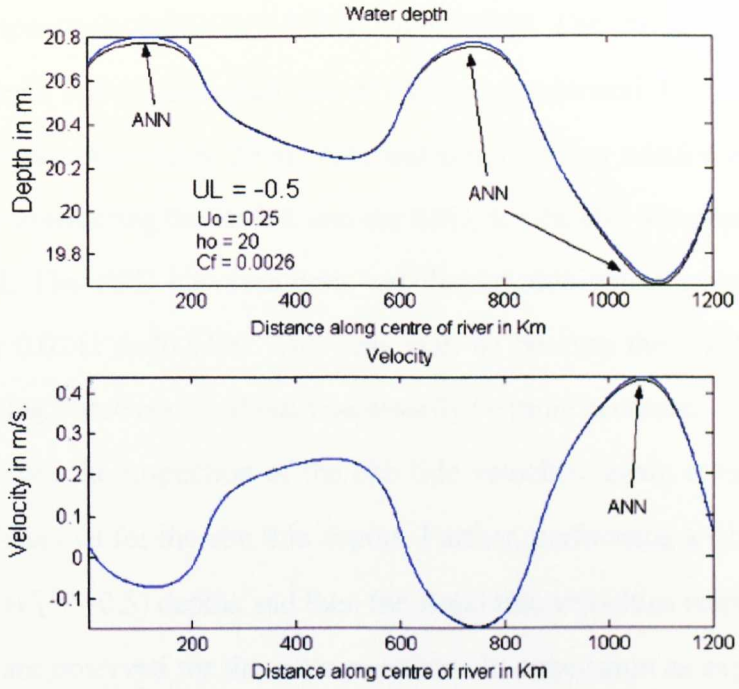

Figure 5.16: 20 m flood tide depth and velocity simulations

With the downstream velocity U_L changed to a flood value of -0.5 m/s, the simulation results are depicted in fig. 5.16.

As can be seen from equation **(3.25)**, as the depth increases, the effects of the bottom friction diminish. Hence if the bottom friction tends to 'dampen' out any errors in the neural network, then the simulation for a 20 m depth would be less accurate than that for a 15 m depth. The converse would be true for a 10 m value.

h_0	U_0	U_L		ME	RMSE	ESD	RPE
20	0.25	0.5	Depth	0.00820	0.01160	0.00050	0.0400
			Velocity	0.00110	0.00180	0.00005	0.0053
		-0.5	Depth	0.01110	0.01420	0.00067	0.0545
			Velocity	0.00160	0.00300	0.00011	0.0079
15	0.25	0.5	Depth	0.00370	0.00420	0.00017	0.0241
			Velocity	0.00066	0.00120	0.00003	0.0044
		-0.5	Depth	0.00520	0.00710	0.00024	0.0345
			Velocity	0.00100	0.00190	0.00006	0.0066
10	0.25	0.5	Depth	0.00360	0.00380	0.00008	0.0350
			Velocity	0.00041	0.00057	0.00002	0.0041
		-0.5	Depth	0.00490	0.00620	0.00021	0.0479
			Velocity	0.00057	0.00096	0.00004	0.0056

Table 5.21: Comparison of errors for different depths

In comparing the ebb tide (U_L = 0.5) data for the depths, from 10 m through to 20 m, the ME values are respectively 0.00360, 0.00370 and 0.00820. The ME is obviously increasing with increasing depth and the attendant loss of friction as expected. It is also noticeable that the greatest loss occurs from 15 to 20 m depth and is not a linear relationship. Continuing in a similar manner, considering the RMSE and the ESD, for the ebb tide depths, a very similar trend is observed. The RPE however does not display this consistent trend since it has values of 0.0350, 0.0241 and0.0400. This may well be because the middle value (0.0241) relates to the training phase and so should necessarily be more accurate.

In a likewise fashion, the inspection of the ebb tide velocities again reveals (including the RPE) a trend such as that for the ebb tide depths. Further, performing a similar analysis first for the flood tide (U_L = -0.5) depths and then the flood tide velocities respectively, the same consistent trends are observed for the error measures. In conclusion as expected, apart from the RPE for the flood tide depths, the error measures increase consistently in accordance with the increase in still water depth.

5.5.2 Variation of the coefficient of bottom friction

Here the intention was to see the effect of changing the friction coefficient C_f on the neural network simulation. The value was decreased from the training value of 0.0026 to 0.0015, 0.0010 and finally 0.0000 respectively. The last value would of course result in the zero friction model of Chapter 4 but for the fact that the neural network is now employing different weights and a different activation function. Plots of the simulations for 0.0015, 0.0010 and 0.0000 follow (figs. 5.17 to 5.22). The simulations were performed as usual, for both an ebb tide, U_L = 0.5 m/s and a flood tide, U_L = -0.5 m/s.

Ebb tide: coefficient = 0.0015

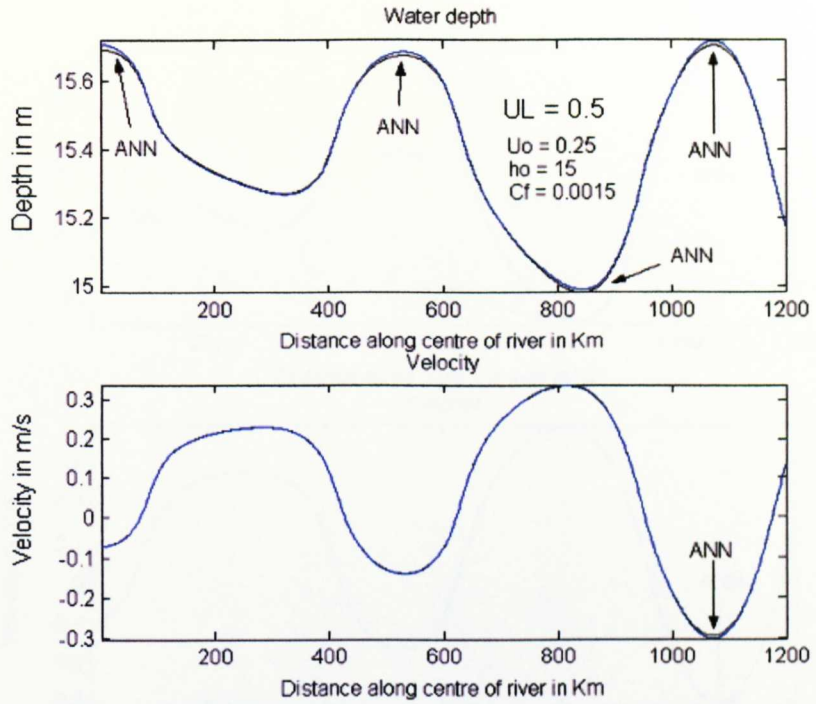

Figure 5.17: Ebb tide depth and velocity simulations with 0.0015 coefficient

Flood tide: coefficient = 0.0015

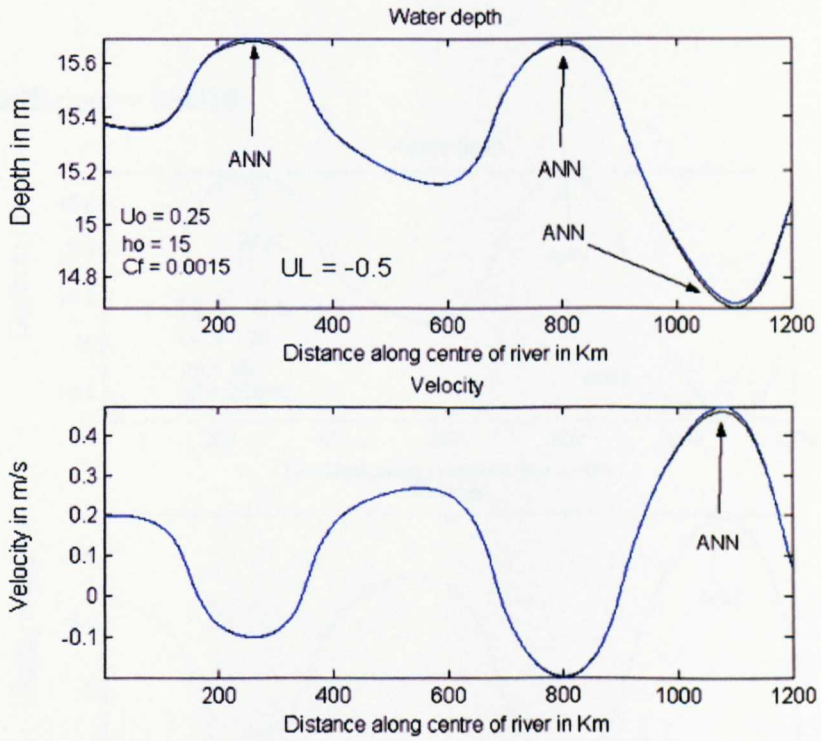

Figure 5.18: Flood tide depth and velocity simulations with 0.0015 coefficient

Ebb tide: coefficient = 0.0010

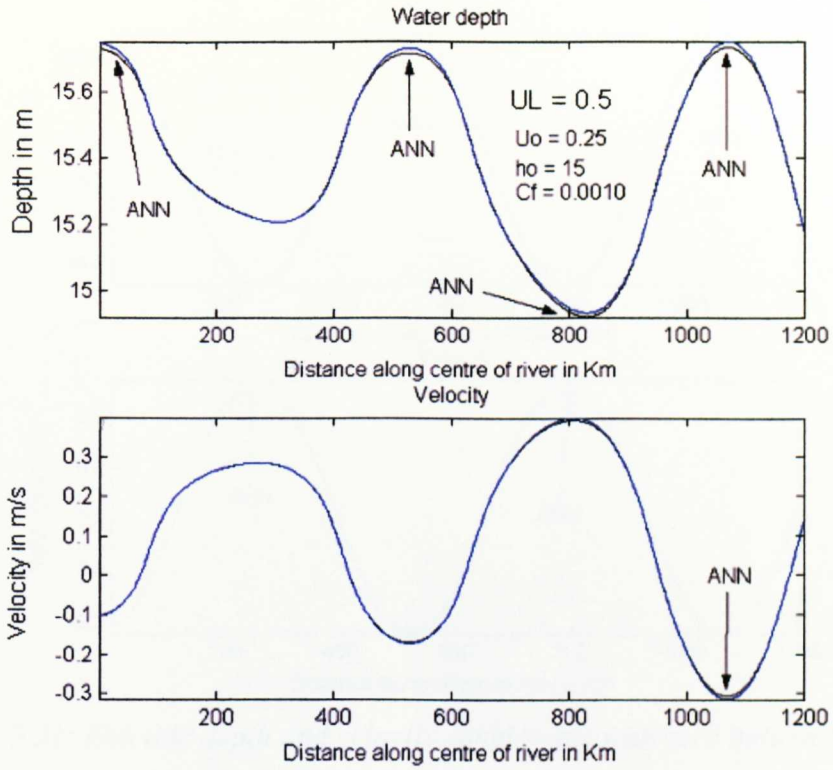

Figure 5.19: Ebb tide depth and velocity simulations with 0.0010 coefficient

Flood tide: coefficient = 0.0010

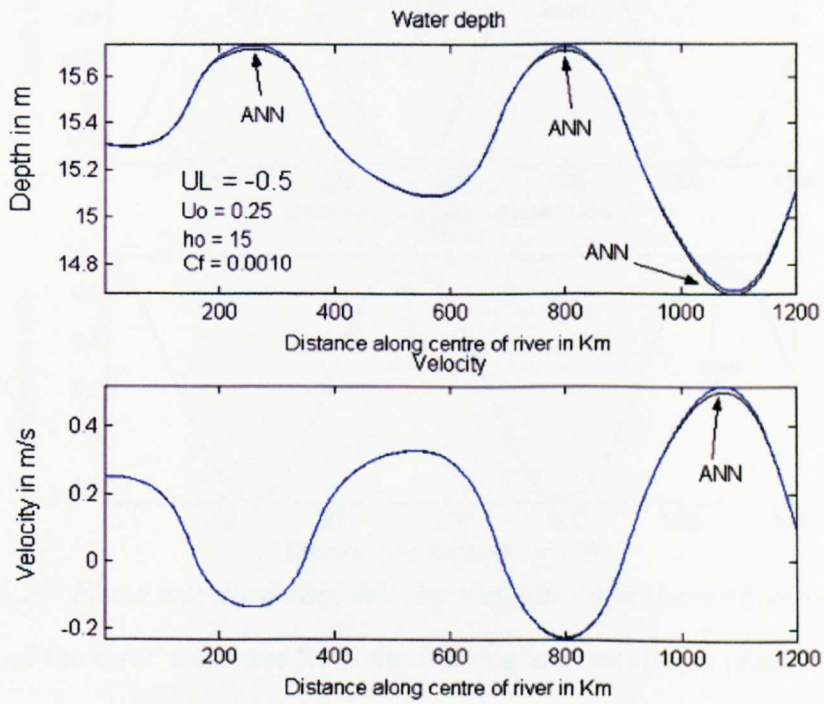

Figure 5.20: Flood tide depth and velocity simulations with 0.0010 coefficient

Ebb tide: coefficient = 0

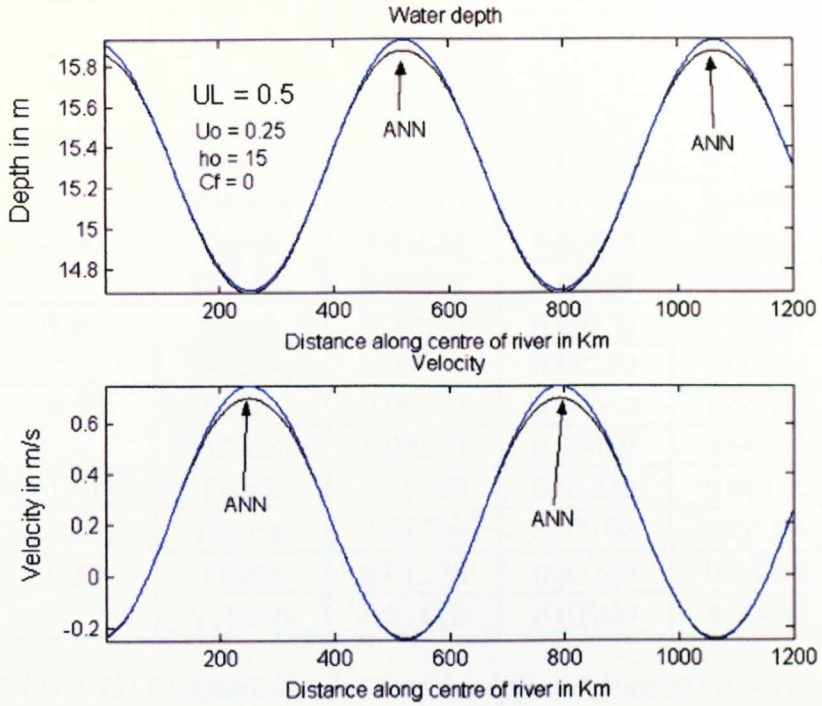

Figure 5.21: Ebb tide depth and velocity simulations with zero bottom friction

Flood tide: coefficient = 0

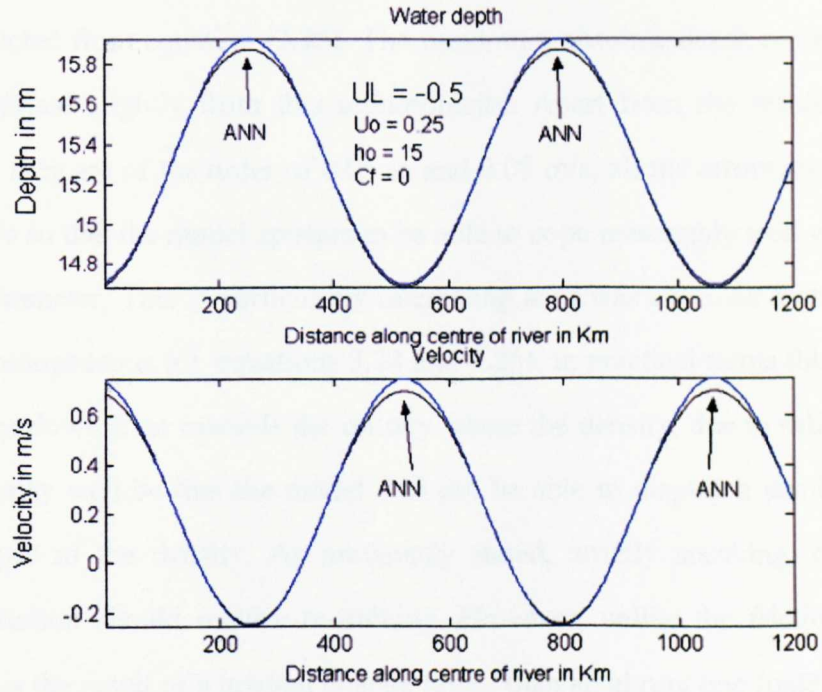

Figure 5.22: Flood tide depth and velocity simulations with zero bottom friction

The inclusion of the error measures from the training and validation phases (cf. figs. 5.9 to 5.12) results in the following table of errors:

C_f	U_L		ME	RMSE	ESD	RPE
0.0026	0.50	Depth	0.00370	0.00420	0.00017	0.0241
		Velocity	0.00066	0.00120	0.00003	0.0044
	-0.50	Depth	0.00520	0.00710	0.00024	0.0345
		Velocity	0.00100	0.00190	0.00006	0.0066
0.0015	0.50	Depth	0.00520	0.00630	0.00028	0.0336
		Velocity	0.00110	0.00180	0.00005	0.0072
	-0.50	Depth	0.00620	0.00810	0.00036	0.0407
		Velocity	0.00170	0.00340	0.00012	0.0109
0.0010	0.50	Depth	0.00630	0.00830	0.00037	0.0408
		Velocity	0.00160	0.00260	0.00008	0.0103
	-0.50	Depth	0.00740	0.00940	0.00043	0.0483
		Velocity	0.00240	0.00480	0.00017	0.0160
0	0.50	Depth	0.01660	0.02280	0.00110	0.1072
		Velocity	0.01250	0.02100	0.00084	0.0801
	-0.50	Depth	0.01570	0.02160	0.00099	0.1017
		Velocity	0.01440	0.02260	0.00094	0.0924

Table 5.22: Comparison of errors for different friction coefficient

It can be seen that apart from the depth errors for the flood tide, all the other errors are monotonically decreasing with increasing value of the coefficient of friction. This is as would be expected from equation **(3.25)**. The maximum absolute depth errors for the flood simulations deviate slightly from this monotonicity. Apart from the results for the zero friction where they are of the order of 0.05 m and 0.05 m/s, all the errors are less than 0.02 m and 0.02 m/s so that the model appears to be able to cope reasonably well with changes in the friction parameter. This is particularly interesting as it was assumed that the density of the water is homogeneous (cf. equations 3.24 and 3.25). In practical terms this would not be the case further down river towards the estuary where the density, due to salinity, would be increasing. It may well be that the model will not be able to display a similar 'flexibility' towards changes in the density. As previously stated, strictly speaking, changes to the density and friction should require re-training. However, unlike the friction, the salinity concentration is the result of a gradual change rather than an abrupt one (unless a salt wedge is present on the flood tide) and so it would be impractical to continually keep retraining the system to represent different locations in the river.

5.5.3 Variation of the downstream (estuarine) velocity

Keeping the forcing upstream (river end) velocity at its value as used in both training and validation, that is, 0.25 m/s, the coefficient of friction at 0.0026 and the still water depth at 15 m, three simulations were conducted involving different downstream (tidal) velocities from those used in the training and validation phases. The downstream estuarine velocities used in the training and validation were 0.5 and -0.5 m/s respectively The first new downstream value is 0.25 m/s which corresponds to an ebb flow. The second value for simulation is -0.25 m/s which represents a flood tide of exactly the same magnitude as the first one but now in the opposite direction. The third test velocity is zero. That is, the tide is 'on the turn'. Plots of these three simulations as well as a table of the error measure results follow:

Ebb tide: $U_L = 0.25$

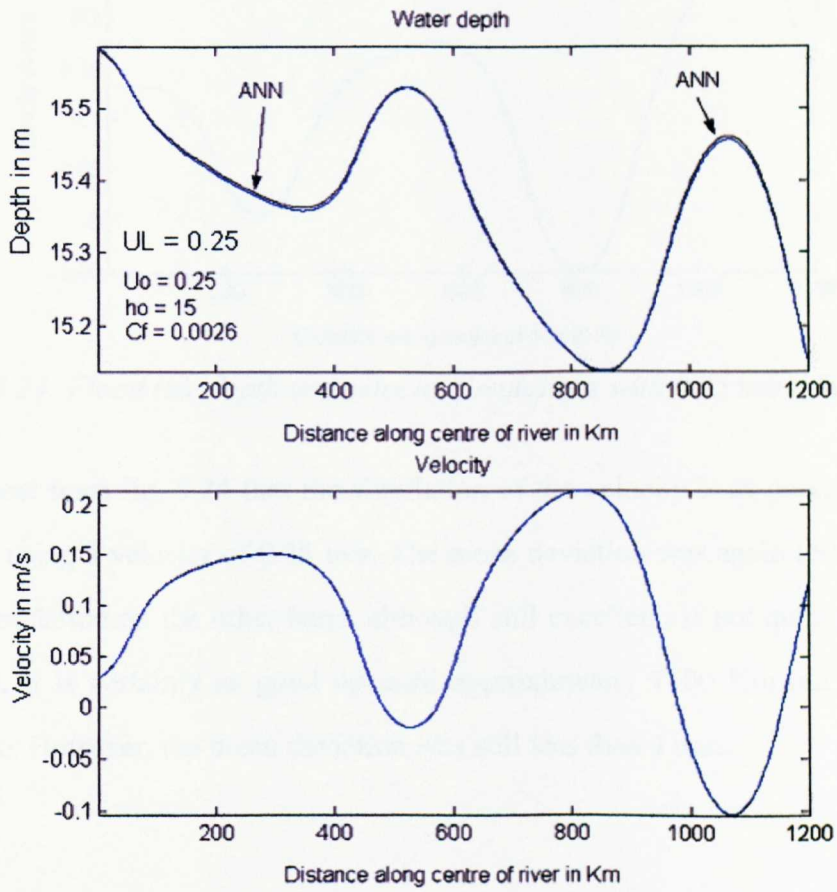

Figure 5.23: Ebb tide depth and velocity simulations with 0.25 m/s tidal velocity

Fig. 5.23 demonstrates that the network is able to cope extremely well with this change to the downstream velocity. The mean deviation between the target depth and the ANN

simulation was less than 3 mm. The agreement between the target velocity and the ANN result would appear to be even better. In fact, the mean deviation was less than 1 mm/s.

Flood tide: U_L = -0.25

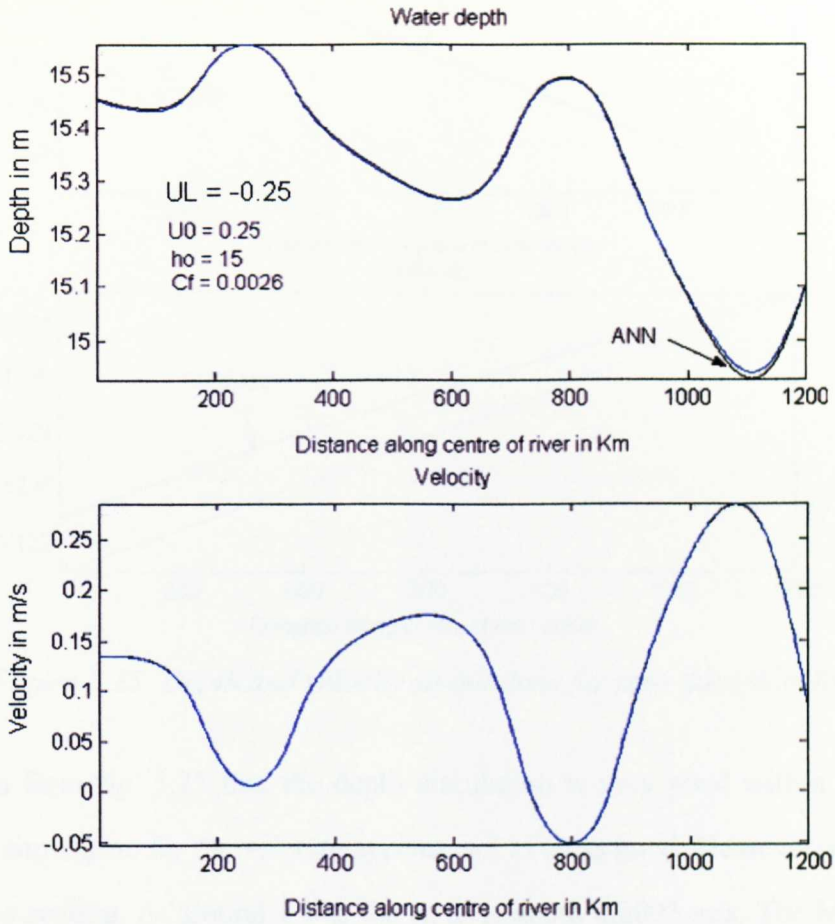

Figure 5.24: Flood tide depth and velocity simulations with 0.25 m/s tidal velocity

It would appear from fig. 5.24 that the simulation of the velocity is as good as that for the previous test using a velocity of 0.25 m/s. The mean deviation was again less than 1 mm/s. The depth simulation on the other hand, although still excellent, is not quite as good as the previous test. It is certainly as good up until approximately 1100 Km but then becomes slightly worse. However, the mean deviation was still less than 4 mm.

Zero tidal flow: $U_L = 0$

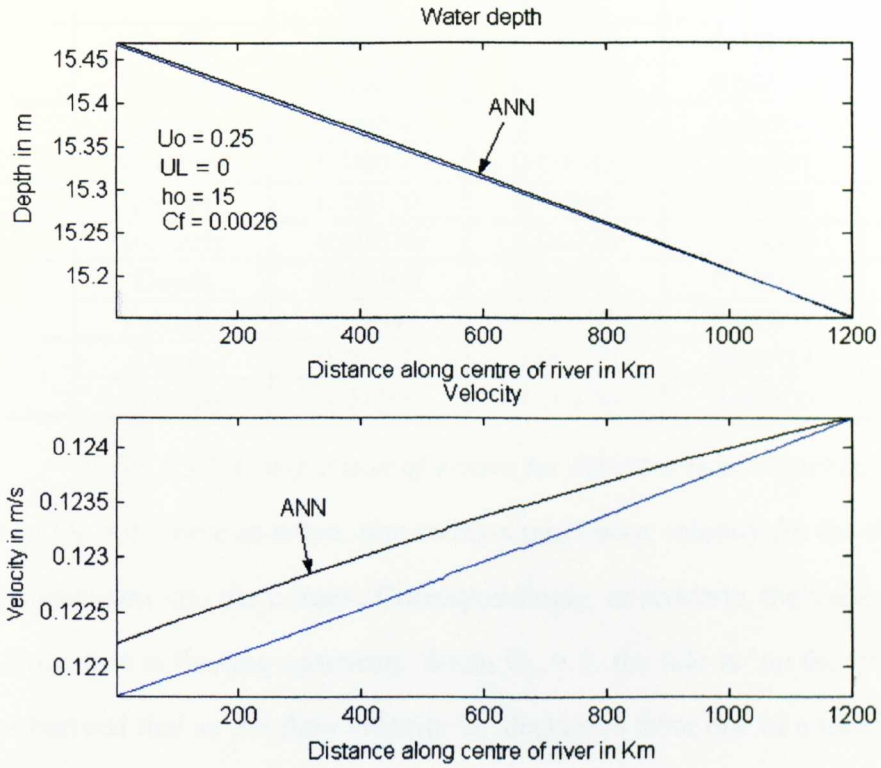

Figure 5.25: Depth and velocity simulations for zero tidal velocity

It can be seen from fig. 5.25 that the depth simulation is very good with a ME less than 0.003 m. The simulation for the velocity appears not to be as good. However, account has to be taken of the scaling. At around 1 Km the error is about 0.0005 m/s. The ME in fact was less than 0.0004 m/s. The velocity simulation also demonstrates the immediate effects of bottom friction on the forcing velocity close to 0 Km by almost halving it. If the error measure values for the simulations used for training and validation are included, that is when the estuarine velocities were 0.5 and -0.5 respectively, and are collated into a table, a clearer picture emerges as to the effect of this estuarine velocity on the simulations:

U_L		ME	RMSE	ESD	RPE
0.5	Depth	0.00370	0.00420	0.00017	0.0241
	Velocity	0.00066	0.00120	0.00003	0.0044
0.25	Depth	0.00290	0.00320	0.00006	0.0188
	Velocity	0.00036	0.00040	0.00001	0.0024
0	Depth	0.00270	0.00300	0.00004	0.0175
	Velocity	0.00036	0.00038	0.00000	0.0023
-0.25	Depth	0.00390	0.00470	0.00017	0.0256
	Velocity	0.00041	0.00051	0.00002	0.0027
-0.5	Depth	0.00520	0.00710	0.00024	0.0345
	Velocity	0.00100	0.00190	0.00006	0.0066

Table 5.23: Comparison of errors for different tidal velocities

The value of $U_L = 0.5$ here as noted, represents a maximum velocity for the ebb flow, that is the flow downstream into the estuary. Correspondingly, to reiterate, the value of $U_L = -0.5$ is the flood flow, that is flowing upstream. When $U_L = 0$, the tide is 'on the turn'. From table 5.23, it is observed that as the flow velocity U_L decreases from one of a maximum ebb flow down to a value of 0, the four error measures also decrease in value. Similarly, as the flow now becomes a flood flow and 'decreases' to a value of -0.5, so the error measures once again increase. This behaviour is, intuitively, to be expected. Also, it is noticeable that the values of these measures when $U_L = 0.5$ are all less than their counterparts when $U_L = -0.5$. This might possibly be explained by the fact that the upstream velocity U_0 is combining with the flood flow and so is susceptible to more errors. That is, U_0 and U_L are now opposing each other and so the generated combined flow is more 'erratic' making it more difficult for the neural network to simulate it as accurately.

5.5.4 Variation of the upstream velocity

Unlike the forcing (estuarine) velocity U_L which is tidal and sinusoidal (cf. Chapter 3), the upstream velocity U_0 is the result of the input from various tributaries, groundwater and flow through various gauge stations. Although subject to fluctuations as the result of seasonal effects, over short periods of time, it does not vary significantly, unlike U_L which decreases and then reverses every half tidal period. The training and validation values for U_0, that is 0.25 m/s, are reasonable (but assumed) estimates for this flow. Further, this is the value used in the model of Johns [50] and so it was readily available for comparison. More

simulations were performed in four different scenarios (other than the training and validation sessions) using two new values for the upstream velocity U_0. Investigations were conducted first for both the flood tide and the ebb tide regimes with a value of 0.1 for U_0 and then secondly for the same two regimes but now with a value of 0.4 for U_0. These two different values, 0.1 and 0.4, could hypothetically represent the different seasonal flows.

Ebb tide: $U_0 = 0.10$

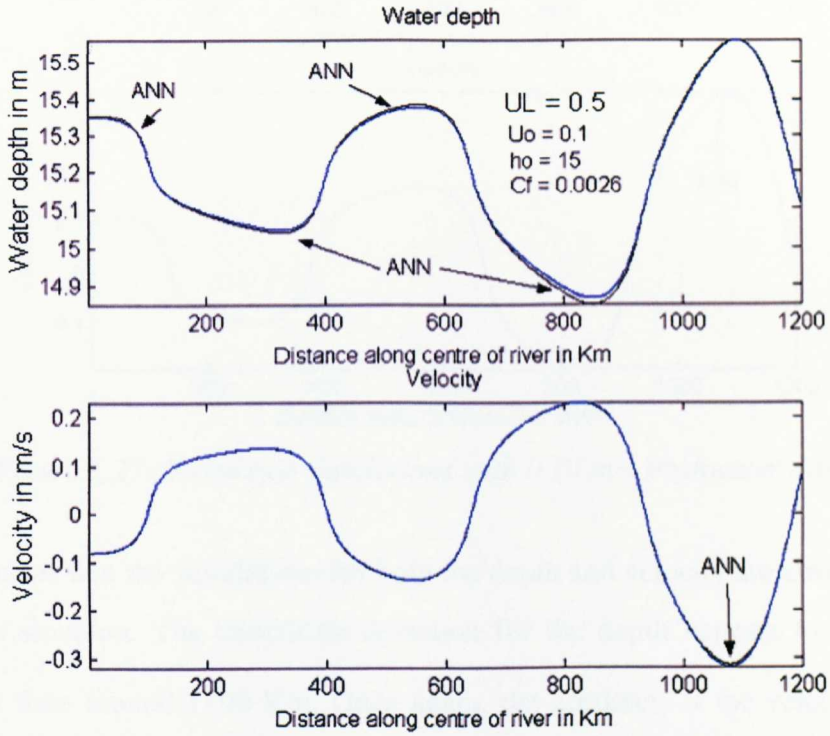

Figure 5.26: Ebb tide simulations with 0.10 m/s freshwater flow

From fig. 5.26 it would seem that the maximum deviation for the depth is $O(0.02)$ m around 850 Km but apart from that, the accuracy of the simulation is extremely good with a mean absolute error of less than 0.006 m. The simulation of the velocity is even better with a mean absolute error less than 0.002 m.

Flood tide: $U_0 = 0.10$

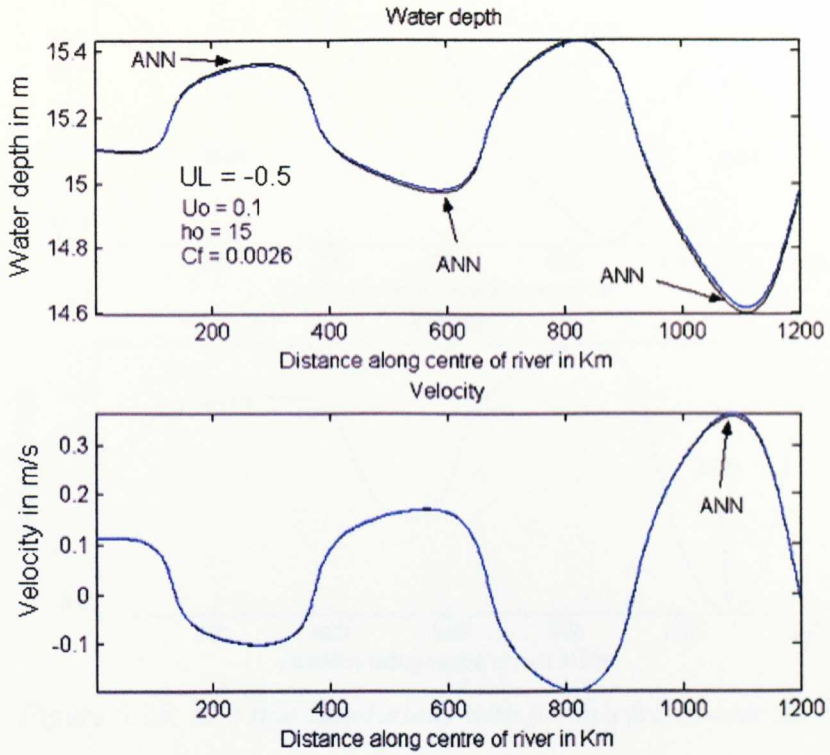

Figure 5.27: Flood tide simulations with 0.10 m/s freshwater flow

Fig. 5.27 indicates that the simulations for both the depth and velocity are also very good for the flood tide situation. The maximum deviation for the depth appears to be $O(0.02)$ m again but this time around 1100 Km. Once again, the accuracy of the velocity simulation appears to be better than the depth. The mean absolute error of the depth simulation was less than 0.008 m whilst that for the velocity, as it was for the ebb tide regime, was less than 0.002 m.

Ebb tide: $U_0 = 0.4$

In the previous two tests, the upstream forcing velocity U_o was much smaller than the forcing tidal velocity U_L and so it is to be expected that the tidal term would have much greater influence on the flow and hence the accuracy of the ANN estimations. Intuitively therefore, increasing the upstream velocity to near that of the downstream one should give rise to a completely different scenario with resultant effects on the ANN estimations.

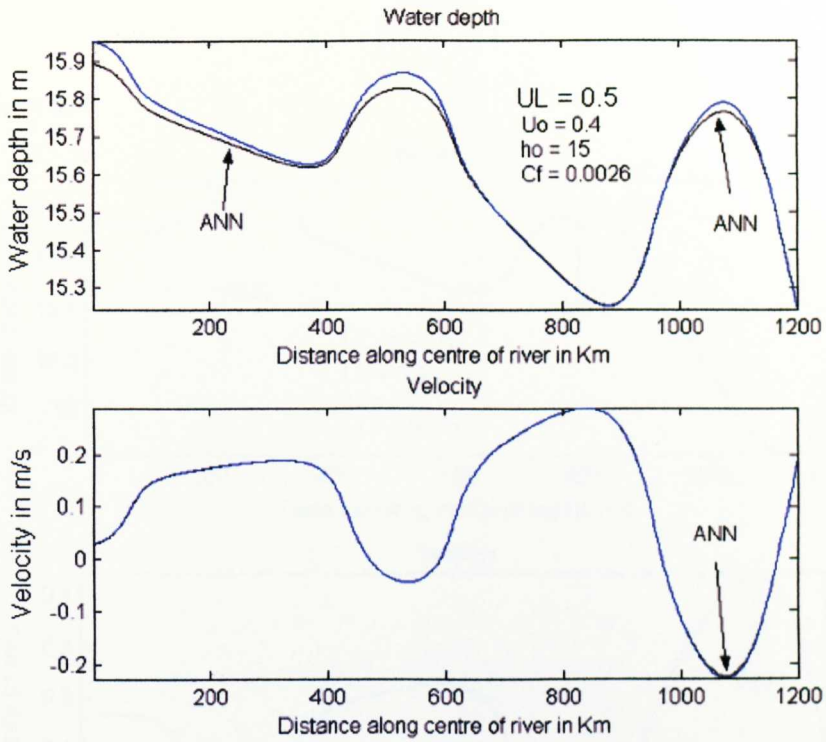

Figure 5.28: Ebb tide simulations with 0.4 m/s freshwater flow

The depth simulation as depicted in fig. 5.28 has three significant deviations in the estimation of the depth, at around 1 Km, 550 Km and 1050 Km although in reality, they are fairly small being of $O(0.07)$, $O(0.05)$ and $O(0.03)$ m respectively. The mean absolute error was less than 0.02 m. The velocity simulation by contrast is very much better with a mean absolute error less than 0.001m, which is interesting, as any errors in the depth are propagated into the velocity network as noted earlier.

Flood tide: $U_0 = 0.4$

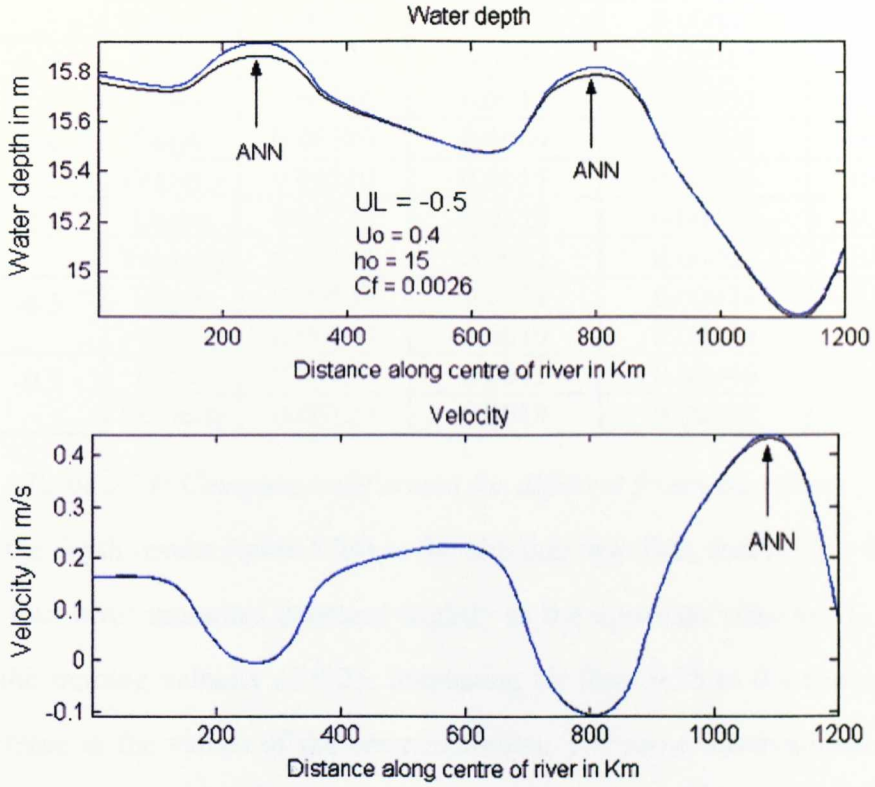

Figure 5.29: Flood tide simulations with 0.4 m/s freshwater flow

From fig. 5.29 it would appear that the depth simulation has two significant deviations in the depth estimation at around 250 Km and 800 Km. However they are not particularly large being of $O(0.05)$ and $O(0.03)$ m respectively. The mean absolute error was again less than 0.02 m. The velocity simulation as in the previous test, is very much better with a mean absolute error less than 0.002 m.

Collating the error measure results, including those obtained during the course of training and validation, leads to the following table:

U_0	U_L		ME	RMSE	ESD	RPE
0.4	0.5	Depth	0.01690	0.0228	0.00110	0.1070
		Velocity	0.00057	0.0011	0.00003	0.0038
0.25	0.5	Depth	0.00370	0.0042	0.00017	0.0241
		Velocity	0.00066	0.0012	0.00003	0.0044
0.1	0.5	Depth	0.00580	0.0069	0.00029	0.0387
		Velocity	0.00110	0.0015	0.00006	0.0071
0.4	-0.5	Depth	0.01720	0.0228	0.00110	0.1100
		Velocity	0.00110	0.0022	0.00008	0.0070
0.25	-0.5	Depth	0.00520	0.0071	0.00024	0.0345
		Velocity	0.00100	0.0019	0.00006	0.0066
0.1	-0.5	Depth	0.00750	0.0091	0.00040	0.0504
		Velocity	0.00120	0.0019	0.00008	0.0078

Table 5.24: Comparison of errors for different freshwater flows

Considering the depth results (table 5.24) in the ebb tide case first, that is $U_L = 0.5$, it can be seen that all four error measures decrease slightly as the upstream velocity U_0 is increased from 0.1 to the training velocity of 0.25. Increasing U_0 from 0.25 to 0.4 shows a dramatic five fold increase in the values of the error measures. The same observations apply to the depth results in the flood tide case, $U_L = -0.5$, where increasing U_0 beyond 0.25 results in a three to five fold increase in the values of these error measures.

Performing a similar inspection of the velocity results firstly in the ebb tide case, then apart from the RMSE result, there is a small but noticeable decrease in their values when increasing from $U_0 = 0.1$ to $U_0 = 0.25$. Thereafter, there is no significant change. For the flood tide situation, the same trend prevails but to a smaller extent.

Recalling that the training value for U_0 was 0.25, it would appear that any decrease in this value has a much larger influence on the ability of the neural network to accurately simulate the depth than an increase beyond this training value. The same argument applies to the velocity but is of a much smaller nature.

5.5.5 Variation of both upstream and downstream (estuarine) velocities

The scenario here is a more complicated situation to that in the previous test. In these tests, there will be a variation of the upstream velocity U_0 from its training value of 0.25 as well

as changing the value of the downstream (estuarine) value U_L from its training and validation values of 0.5 and -0.5.

Ebb tide: $U_0 = 0.4$, $U_L = 0.7$

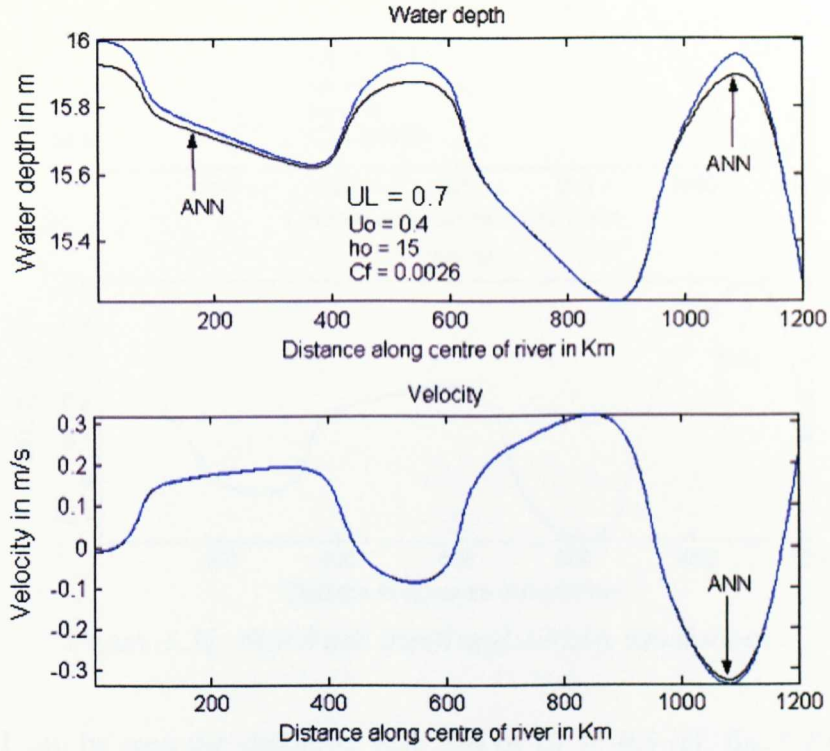

Figure 5.30: Ebb tide depth and velocity simulations

The depth simulation as depicted in fig. 5.30 is very similar to that with $U_L = 0.5$ (cf. fig. 5.28). Again, there are three significant deviations in the estimation of the depth, namely at around 1 Km, 550 Km and 1050 Km. In reality, again they are not very large being of $O(0.08)$, $O(0.06)$ and $O(0.06)$ m respectively. The mean absolute error was less than 0.03 m. The velocity simulation again (cf. fig. 5.25), contrastingly is much better having a mean absolute error less than 0.002 m.

Flood tide: $U_0 = 0.4$, $U_L = -0.7$

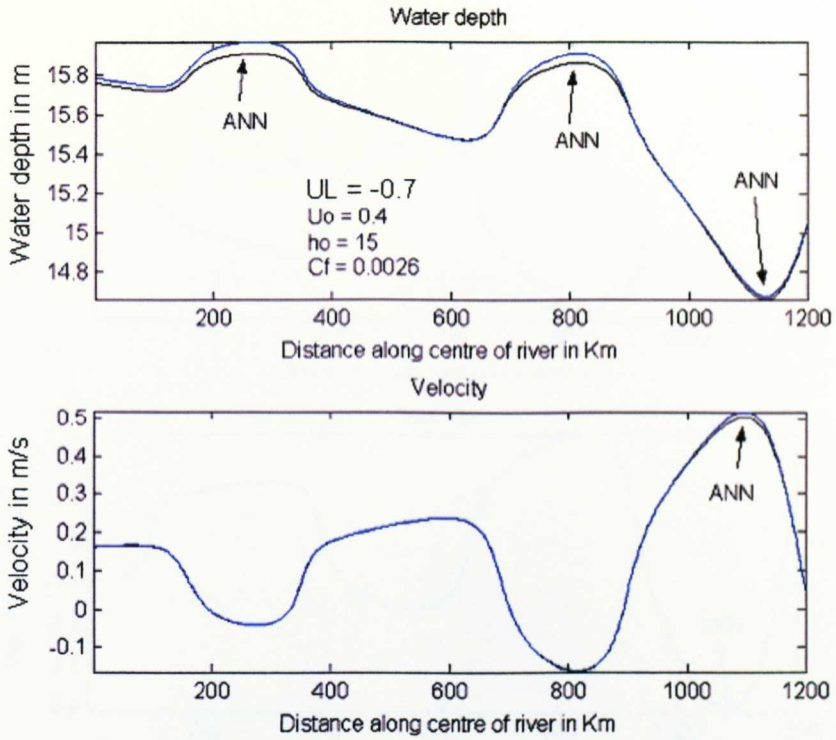

Figure 5.31: Flood tide depth and velocity simulations

From fig. 5.31 can be seen the similarity with that of $U_L = -0.5$ (cf. fig. 5.29). Once again, the depth simulation has two significant deviations in the depth estimation at around 250 Km and 800 Km. However once more they are not particularly large being of $O(0.08)$ and $O(0.05)$m respectively. The mean absolute error was less than 0.03 m. The velocity simulation as in the previous test (cf. 5.29), is very much better with a mean absolute error of less than 0.002m.

Ebb tide: $U_0 = 0.1$, $U_L = 0.7$

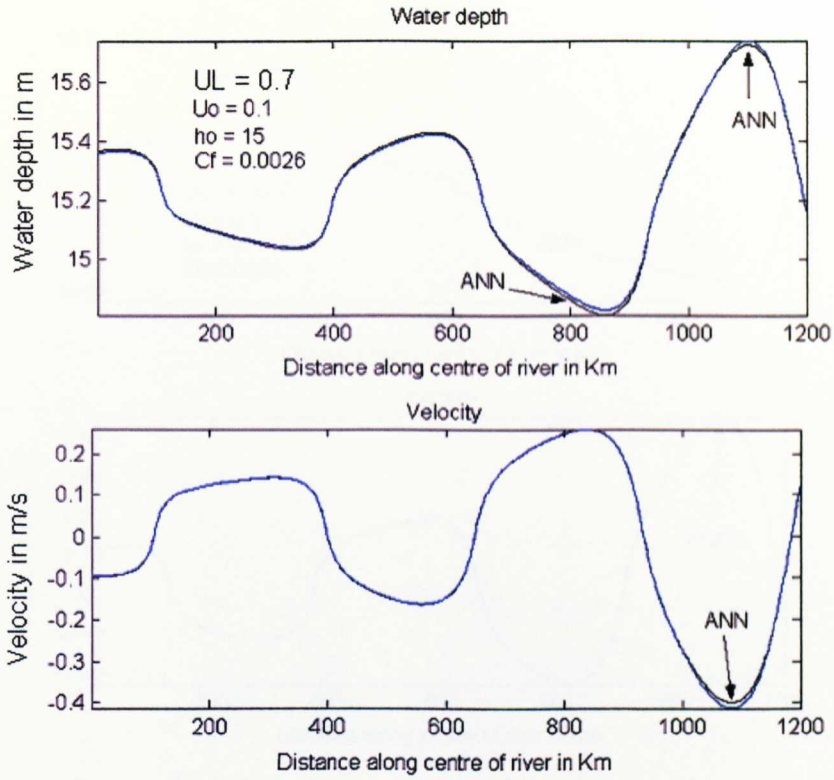

Figure 5.32: Ebb tide depth and velocity simulations

From a close inspection of fig. 5.26, it is possible to see the similarity in the accuracies of both the depth and velocity simulations with that of fig. 5.32. From the latter, the maximum deviation for the depth is $O(0.02)$ m around 850 Km whilst that for the velocity is also of the same order at around 1050 Km. The mean absolute errors were 0.0068 and 0.0018 for the depth and velocity respectively. This contrasts well with that of fig. 5.26 where the errors were 0.0058 and 0.0011.

Flood tide: $U_0 = 0.1$, $U_L = -0.7$

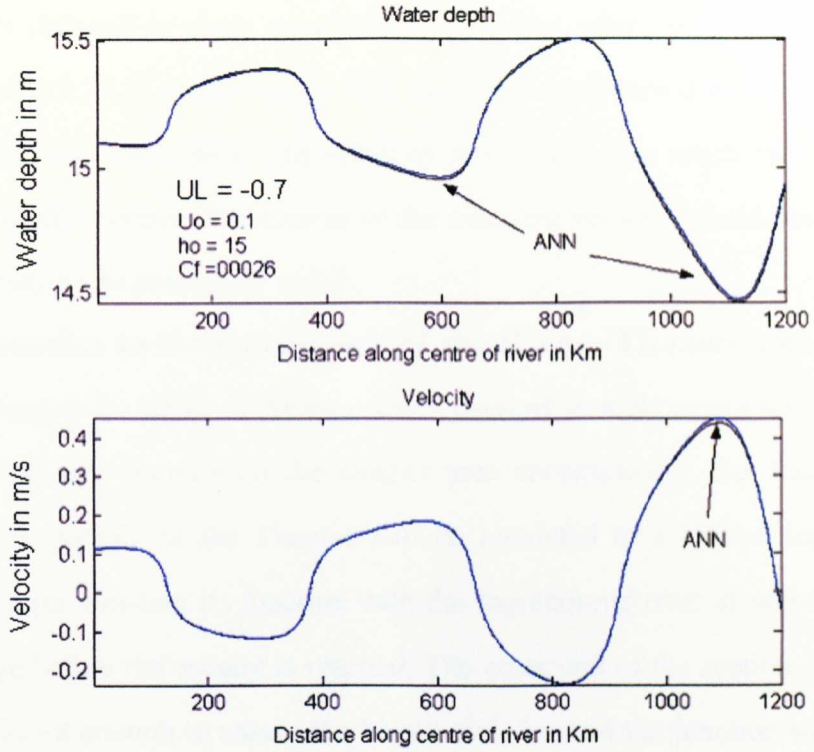

Figure 5.33: Flood tide depth and velocity simulations

The similarity between fig. 5.33 and fig. 5.27 appears to be even greater than that between fig.5.32 and fig. 5.26. The mean absolute errors for the depth and velocity are respectively, 0.0074 and 0.0019 whilst those relating to the simulation in fig. 5.27 were 0.0075 and 0.0012.

U_0	U_L		ME	RMSE	ESD	RPE
0.4	0.7	Depth	0.02390	0.03220	0.00150	0.1508
		Velocity	0.00130	0.00280	0.00008	0.0087
	-0.7	Depth	0.02250	0.02990	0.00140	0.1436
		Velocity	0.00190	0.00420	0.00015	0.0124
0.25	0.5	Depth	0.00370	0.00420	0.00017	0.0241
		Velocity	0.00066	0.00120	0.00003	0.0044
	-0.5	Depth	0.00520	0.00710	0.00024	0.0345
		Velocity	0.00100	0.00190	0.00006	0.0066
0.25	0	Depth	0.00270	0.00300	0.00004	0.0175
		Velocity	0.00036	0.00038	0.00000	0.0023
0.1	0.7	Depth	0.00680	0.00800	0.00036	0.0447
		Velocity	0.00180	0.00370	0.00011	0.0123
	-0.7	Depth	0.00740	0.00860	0.00038	0.0491
		Velocity	0.00190	0.00360	0.00014	0.0122

Table 5.25: Comparison of errors for different U_0 and U_L

Varying both the upstream and the downstream velocities at the same time necessarily makes it more difficult to draw conclusions. However, after some analysis of the error measures in table 5.25, it would appear that the trends as observed in the previous tests, of varying U_0 only, also apply here. The effect of changing U_0 has much more influence than changing U_L which is rather fortuitous since the estuarine velocity would, since it is tidal, be continually changing as previously noted.

5.6 Application to the confluence of the Rivers Thames and Medway

Having investigated the effect of the two neural network models over a length of 1200 Km, the investigation now focuses on the smaller area encompassing the Thames / Medway confluence. The domain of the Thames will be restricted to a section length of 60 Km starting from approximately its junction with the Ingrebourne river at which is located the last river gauge before the estuary is reached. The other end of the section is at an arbitrary point but sufficient enough to ensure the boundary is beyond the junction with the Medway following the arguments of Vreugdenhil [95] (cf. Chapter 3). This location in the Thames (at 60 Km) is also convenient as it is approximately the point at which the Yantlet Dredged Channel in the Thames converges with the Medway Approach Channel (also dredged). Similarly, for the Medway, a section length of 35 Km is considered, from this juncture running through the Medway down to approximately the town of Rochester, a distance of about 25 Km from the confluence of the two rivers. Again, this is an arbitrary but convenient point. This juncture is fully depicted in fig. 5.34 that follows. The last gauge on the Medway before reaching the Thames is located at Teston. Flow data in the form of 'cumsecs' for these two gauge stations, Ingrebourne and Teston, is available from the National Rivers Authority.

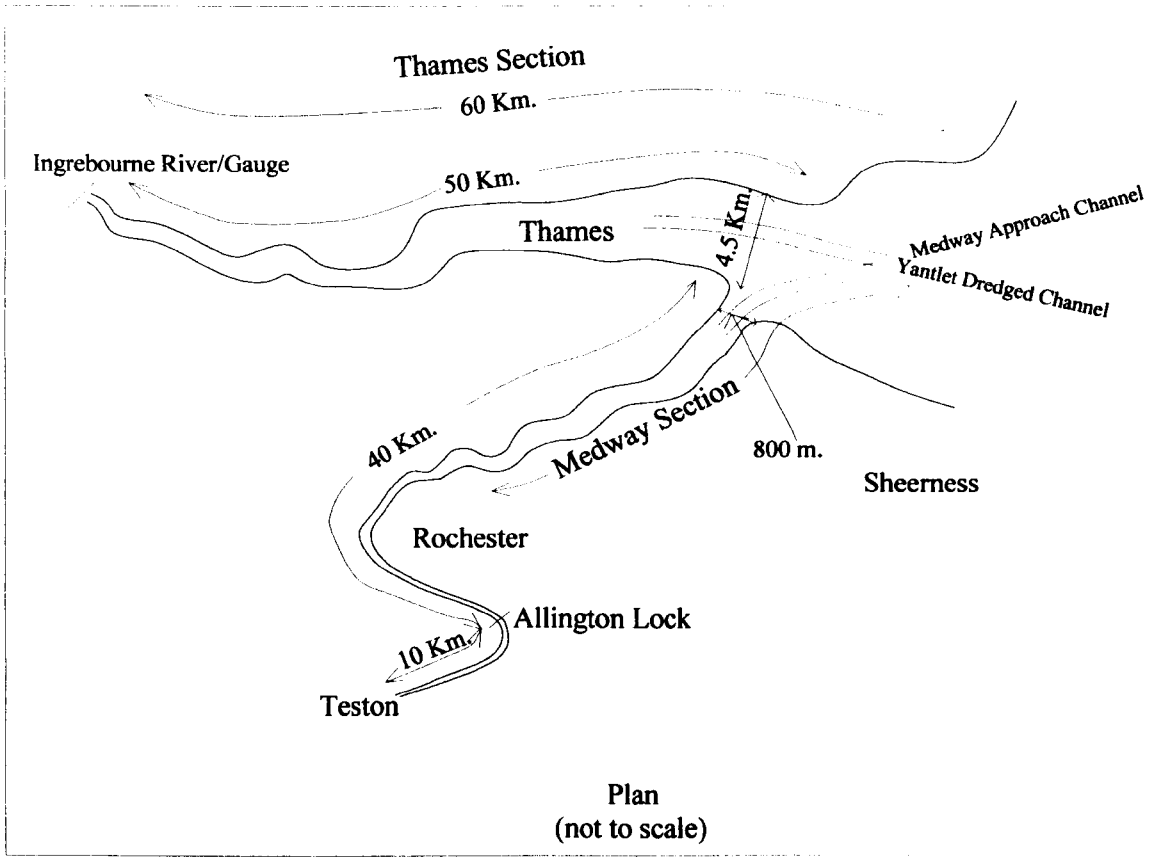

Figure 5.34: Confluence of the Rivers Thames and Medway

Although this research has assumed a model with a constant horizontal bed, it should be noted that in reality, this is of course not true for the two rivers. At Ingrebourne, the depth is about 10m. Proceeding down river towards the estuary a distance of 30 Km from Ingrebourne, the average depth in the main channel of the Thames is around 15 to 20 m. The Medway, likewise, has a depth that increases with a progression down river towards the Thames. Near Rochester the Medway is quite shallow. Proceeding down river from Rochester, its depth increases to about 6.5 m at 13 Km from the mouth reaching 12 m at approximately 6.5 Km from the mouth. At the junction with the Thames itself, the depth is about 20m. However, this said, it was observed (cf. Section 5.5.1) that within limits, the 15 m depth model is quite flexible in terms of variations of the still water depth. A map showing the detailed bathymetry of the area at the confluence of the two rivers is included in Appendix F.

As for the investigations over the 1200 Km-length, simulations were performed using different velocities at both the upstream and downstream ends of the Thames in both flood

tide and ebb tide modes. Six different combinations were applied to the Thames and two to the Medway. In all cases, unlike the investigations in the previous section, the only parameters that were varied were the velocities.

5.6.1 River Thames

The initial data was generated over a 25 tidal cycle period to ensure any disturbance had completely travelled the full length (cf. Chapter 3). Further, keeping the time interval (30 sec.) and the spatial distance (500 m) the same, implied that there would be 61 depth and 60 velocity sections to ensure the same Courant value (cf. Chapter 3). In the simulations that follow, the zero Km-point is at the Ingrebourne river junction. The error measures for all of the simulations are listed in table 5.26 at the end of this section.

Ebb tide: $U_0 = 0.25$, $U_L = 0.5$

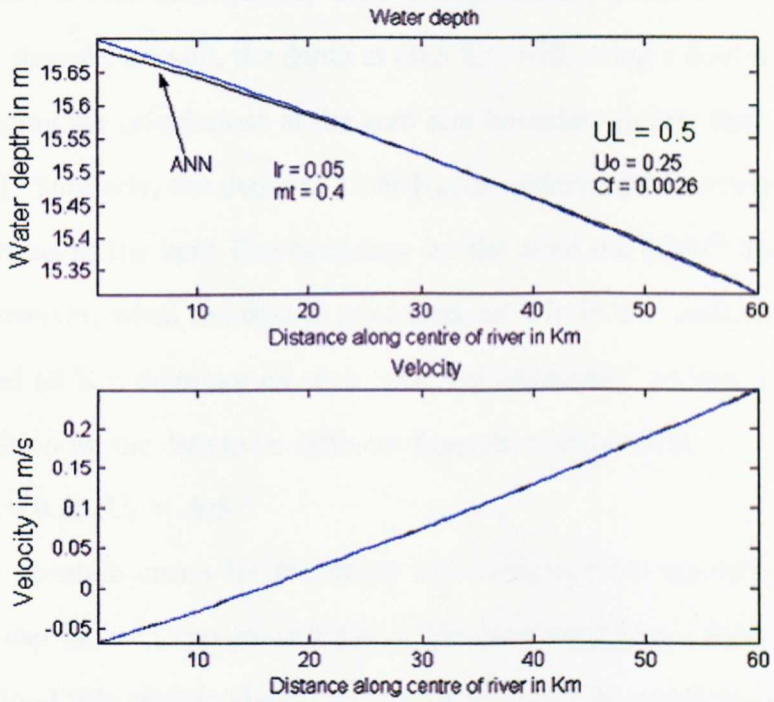

Figure 5.35: Ebb tide simulations in the River Thames

It was noted that the maximum absolute errors for the depth and velocity were 0.0137 m and 0.0040 m/s respectively whilst the error ratios were, for the depth,

$$\frac{ME}{RMSE} \sim \pm 0.741, \quad \frac{ESD}{RMSE} \sim \pm 0.185 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.759$$

and for the velocity:

$$\frac{ME}{RMSE} \sim \pm 0.521, \quad \frac{ESD}{RMSE} \sim \pm 0.128 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 3.407$$

It would appear at first sight that the data generated by the finite difference scheme for this 60 Km-length of the Thames would be the same as that generated for a 1200 Km, the latter being used in the training (cf. Sections 5.4.1 and 5.4.2). Hence, it would be data seen by the networks before and so be useless for any form of testing or validation. However, from an inspection of fig. 3.2 and equations **(3.9)** and **(3.10)**, it can be seen that the depth and velocity are used respectively to calculate the updated data on the next time level and most importantly, in terms of the argument here, the 'influence' progresses both upwards and backwards with progression through the succeeding time levels. In other words, a depth value at 60.5 Km is used to calculate the velocity at 60 Km on the time level above and then this same velocity is then subsequently used to calculate the depth at 59.5 Km on the next time level after that. As a result, the depth at 60.5 Km will, using a double grid interval of 1 Km, be influencing the calculations at the zero Km-boundary in less than 61 iterations, that is time level 61. Similarly, the depth at 1200 Km-boundary will have exerted an influence on the calculations at the zero Km-boundary by the time the 1200^{th} time level has been approached. However, when the data is generated, as it is in this section, in isolation from the river beyond 60 Km there are no such 'external influences' so that 'isolation' here is a sufficient condition for the data to be different from the training data.

Flood tide: $U_0 = 0.25$, $U_L = -0.5$

The maximum absolute errors for the depth and velocity were respectively 0.007 m and 0.0085 m/s. It can be seen that on comparing the maximum errors from fig. 5.35 and fig. 5.36, that the flood tide error is about half that of the ebb tide whilst the opposite is almost exactly the case for the velocity.

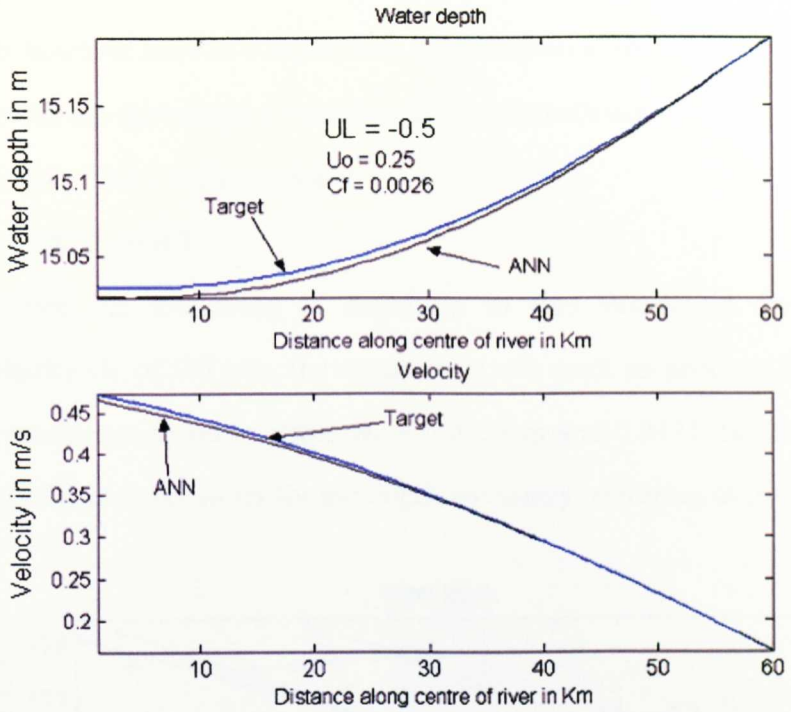

Figure 5.36: Flood tide simulations in the RiverThames

The error measure ratios for the depth and velocity are respectively:

$$\frac{ME}{RMSE} \sim \pm 0.880, \quad \frac{ESD}{RMSE} \sim \pm 0.240 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.800$$

$$\frac{ME}{RMSE} \sim \pm 0.732, \quad \frac{ESD}{RMSE} \sim \pm 0.211 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.780$$

Comparing these ratios with those of the previous (ebb tide) simulation they are about 20 to 60 % larger. This presumably is due to the much better fit of the velocity simulation in fig. 5.35. Either way, the neural network estimates of the data for both flood and ebb scenarios are very good.

Comparing the error measures for the last two simulations (cf. table 5.26) with those of the training (tables 5.10 and 5.13) and validation (tables 5.16 and 5.19) on the full 1200 Km-length, then apart from the ESD results, it was possible to draw the following average comparisons:

$$RMSE_{60} = 1.15 \times RMSE_{1200}$$

$$ME_{60} = 1.35 \times ME_{1200}$$

$$RPE_{60} = 1.33 \times RPE_{1200}$$

The ESD results however have in comparison, been magnified by approximately a factor of seven. These results are encouraging considering that the network is having to simulate with only 5% of the data of the 1200 Km-model.

Ebb tide: $U_0 = 0.25$, $U_L = 0.7$

As expected, since the modelling is departing in this simulation from the training downstream velocity U_L of 0.5 m/s, the accuracy is not quite as good as the two previous scenarios. The maximum absolute errors were 0.0357 m and 0.0121 m/s for the depth and velocity respectively, the maximum for the depth obviously occurring at the zero Km part of the simulation.

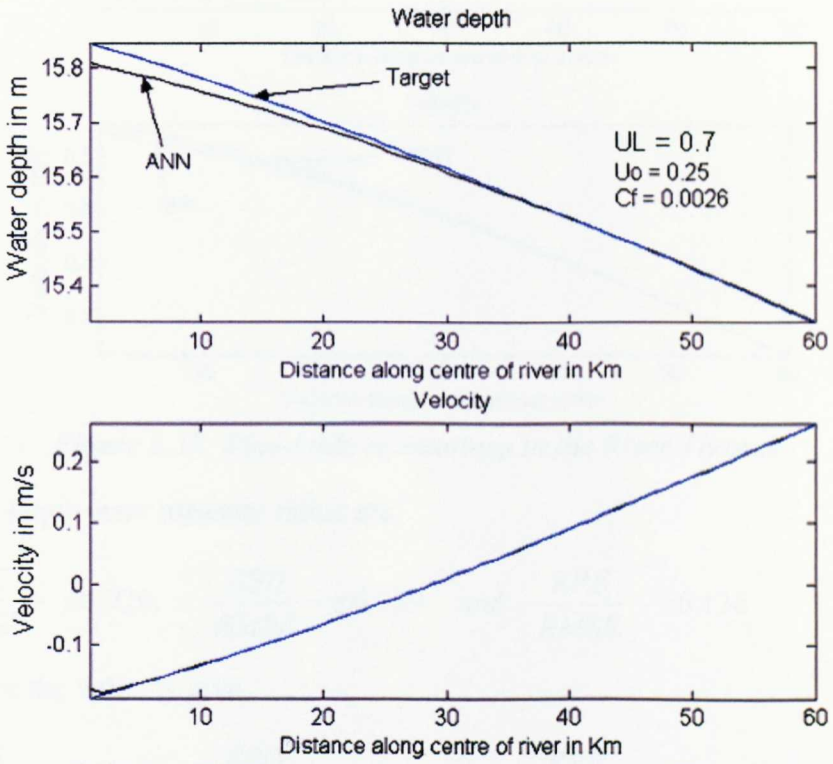

Figure 5.37: Ebb tide simulations in the River Thames

The error measure ratios were, for the depth,

$$\frac{ME}{RMSE} \sim \pm 0.699, \quad \frac{ESD}{RMSE} \sim \pm 0.216 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.431$$

and for the velocity,

$$\frac{ME}{RMSE} \sim \pm 0.374, \quad \frac{ESD}{RMSE} \sim \pm 0.126 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 2.529$$

Flood tide: $U_0 = 0.25$, $U_L = -0.7$

The maximum absolute errors were 0.0114 m and 0.0171 m/s, both occurring towards the beginning of the simulations. The depth error has been reduced by about a factor of three whilst the velocity error has increased marginally from the simulations in the corresponding ebb tide situation.

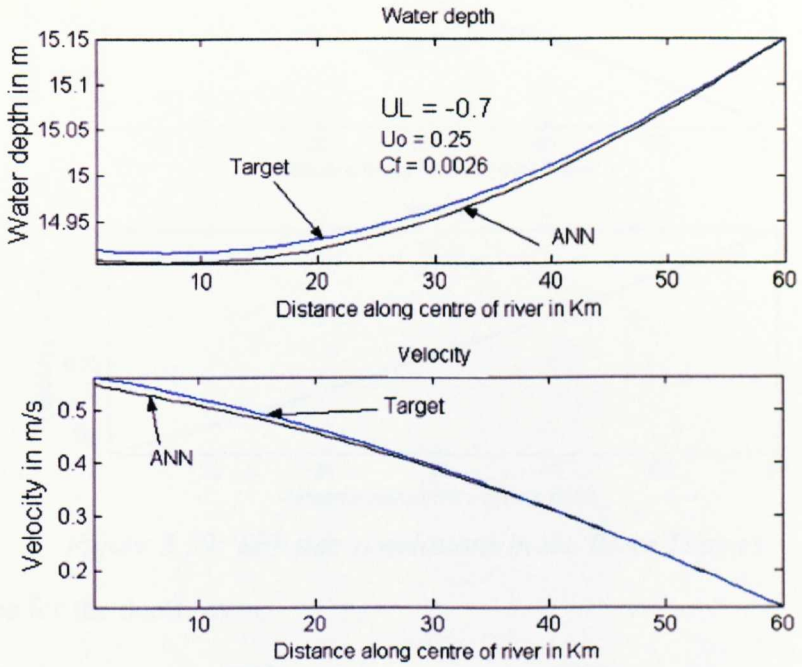

Figure 5.38: Flood tide simulations in the River Thames

As before, the depth error measure ratios are,

$$\frac{ME}{RMSE} \sim \pm 0.920, \quad \frac{ESD}{RMSE} \sim \pm 0.239 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 6.136$$

whilst those for the velocity give,

$$\frac{ME}{RMSE} \sim \pm 0.723, \quad \frac{ESD}{RMSE} \sim \pm 0.205 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.700$$

Comparing the ratios with those of the corresponding ebb tide (fig. 5.37), it was observed that the ratios of the ebb tide have been 'magnified' by between a factor 1.1 and 1.4 for the depth and between 1.6 and 1.9 for the velocity. This degradation of accuracy is easily observed by comparing the velocity simulations of figs. 5.37 and 5.38.

Ebb tide: $U_0 = 0.25$, $U_L = 0.25$

The maximum absolute recorded errors were 0.0032 m and 0.001 m/s for the depth and velocity respectively indicating an extremely good 'fit' as exemplified by fig. 5.39:

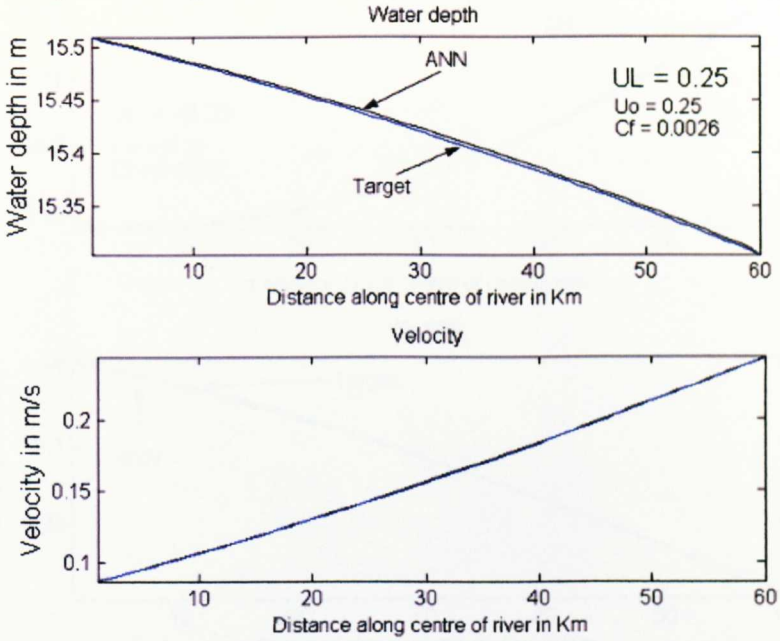

Figure 5.39: Ebb tide simulations in the River Thames

The error ratios for the depth were

$$\frac{ME}{RMSE} \sim \pm 0.962, \quad \frac{ESD}{RMSE} \sim \pm 0.039 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 6.192$$

and for the velocity,

$$\frac{ME}{RMSE} \sim \pm 0.960, \quad \frac{ESD}{RMSE} \sim \pm 0.036 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 6.383$$

Flood tide: $U_0 = 0.25$, $U_L = -0.25$

In this particular simulation the maximum absolute errors were 0.0019 m for the depth and 0.0020 m/s for the velocity. From fig. 5.40, it is observed that most of the depth errors occurred at the beginning and near the end (close to the estuary).

The usual error ratios for depth are

$$\frac{ME}{RMSE} \sim \pm 0.841, \quad \frac{ESD}{RMSE} \sim \pm 0.170 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.568$$

and for the velocity

$$\frac{ME}{RMSE} \sim \pm 0.798, \quad \frac{ESD}{RMSE} \sim \pm 0.202 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.213$$

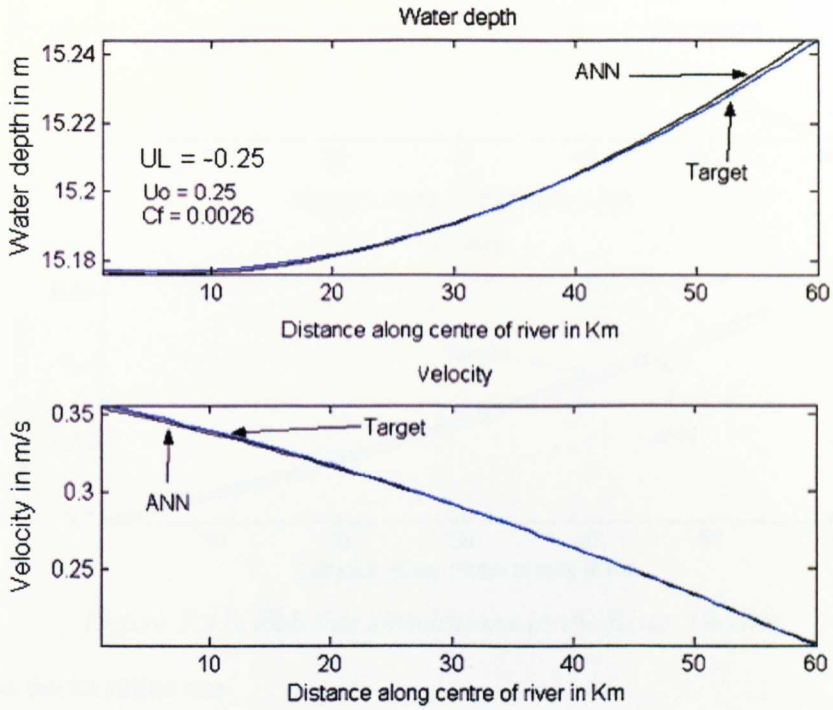

Figure 5.40: Flood tide simulations in the River Thames

From figs. 5.40 and 5.39 it can be seen that the flood tide simulations are of the same order of accuracy as the ebb tide ones. In fact, comparing the error ratios, other than ESD/RMSE, the ratios for the flood tide are approximately 0.8 to 0.9 times those of the ebb tide. However, for the ESD/RMSE ratios, the ebb ratios have been magnified by factors of between 4.2 and 5.7.

Ebb tide: $U_0 = 0.5$, $U_L = 0.25$

In this simulation where the freshwater flow is obviously more dominant than the estuarine flow, the recorded maximum absolute errors were 0.0464 m and 0.01 m/s for depth and velocity. The velocity simulation is very good but the depth neural net estimation is weak at the start of the section. Even so, the difference is only of the order of 0.05m.

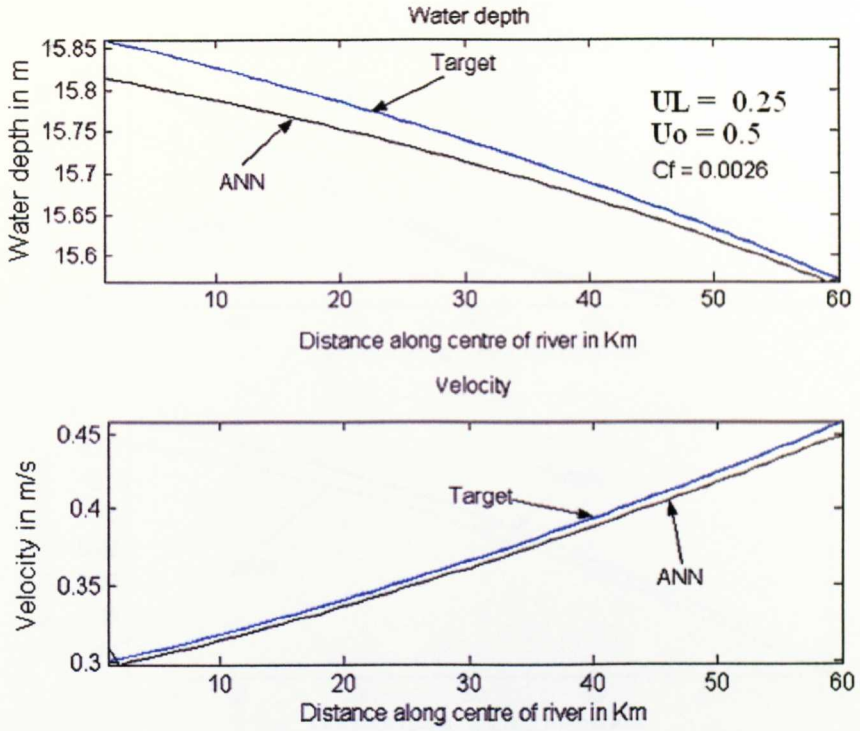

Figure 5.41: Ebb tide simulations in the River Thames

As before, the depth ratios are

$$\frac{ME}{RMSE} \sim \pm 0.904, \quad \frac{ESD}{RMSE} \sim \pm 0.243 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.732$$

and velocity:

$$\frac{ME}{RMSE} \sim \pm 0.963, \quad \frac{ESD}{RMSE} \sim \pm 0.259 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 6.296$$

Flood tide: $U_0 = 0.5$, $U_L = -0.25$

The observed depth and velocity maximum absolute errors were 0.0084 m and 0.0137 m/s.

The simulations produced the depth error ratios

$$\frac{ME}{RMSE} \sim \pm 0.795, \quad \frac{ESD}{RMSE} \sim \pm 0.223 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.154$$

and velocity ratios

$$\frac{ME}{RMSE} \sim \pm 0.946, \quad \frac{ESD}{RMSE} \sim \pm 0.250 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 6.098$$

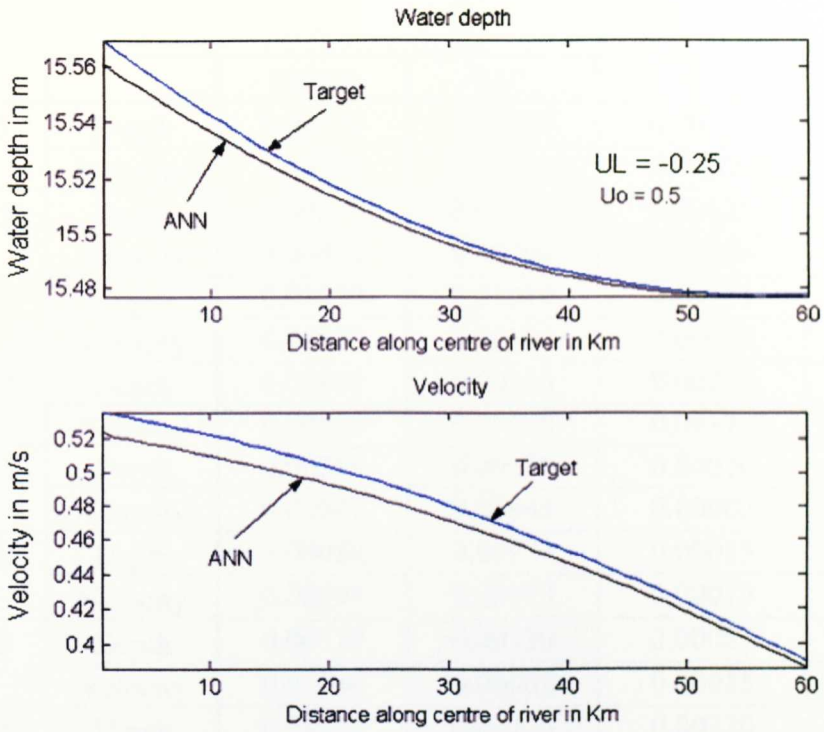

Figure 5.42:Flood tide simulations in the River Thames

Comparing the data relating to fig. 5.41 and fig. 5.42, it is noted that the absolute depth error for the flood tide is about 20% that of the corresponding ebb tide whilst the absolute velocity error for the flood tide is only slightly worse than that of the ebb tide.

In spite of the seemingly rather 'large' error in the depth close to the zero Km boundary (cf. fig. 5.41), the ebb and flood tide simulations are of similar accuracy. In fact, all of the error ratios of the ebb tide have been 'reduced' by factors of between 0.88 and 0.98 in comparison to the flood values.

In addition to the simulations just described, other simulations were performed with U_0 = 0.1 and 0.4. The results are included in table 5.26 overleaf:

U_0	U_L		RMSE	ME	ESD	RPE
0.25	0.5	Depth	0.00540	0.00400	0.00100	0.02570
		Velocity	0.00059	0.00031	0.00007	0.00200
	-0.5	Depth	0.00500	0.00440	0.00120	0.02900
		Velocity	0.00410	0.00300	0.00086	0.01960
	0.7	Depth	0.01530	0.01070	0.00331	0.06780
		Velocity	0.00170	0.00064	0.00022	0.00430
	-0.7	Depth	0.00880	0.00810	0.00210	0.05400
		Velocity	0.00830	0.00600	0.00170	0.03900
	0.25	Depth	0.00260	0.00250	0.00010	0.01610
		Velocity	0.00047	0.00045	0.00002	0.00300
	-0.25	Depth	0.00088	0.00074	0.00015	0.00490
		Velocity	0.00094	0.00075	0.00019	0.00490
0.1	0.5	Depth	0.00370	0.00330	0.00027	0.02150
		Velocity	0.00064	0.00058	0.00015	0.00390
	-0.5	Depth	0.01290	0.01230	0.00330	0.08300
		Velocity	0.00210	0.00170	0.00048	0.01120
0.4	0.5	Depth	0.02550	0.02110	0.00580	0.13380
		Velocity	0.00230	0.00170	0.00047	0.01130
	-0.5	Depth	0.00170	0.00140	0.00012	0.00890
		Velocity	0.00910	0.00740	0.00210	0.04770
0.5	0.25	Depth	0.02800	0.02530	0.00680	0.16050
		Velocity	0.00540	0.00520	0.00140	0.03400
	-0.25	Depth	0.00390	0.00310	0.00087	0.02010
		Velocity	0.00920	0.00870	0.00230	0.05610

Table 5.26: Comparison of simulation errors for River Thames

It was observed that on plotting the four error measures respectively for the ebb depths from table 5.26, they all demonstrated an identical type of line graph with the RPE showing the largest monotonic increase. Overall, the best results (smallest error) occurred when $U_0 = U_L = 0.25$ m/s. The worst results were when U_0 was close to 0.5 m/s. Inconsistent patterns were noted for the flood depths, the best results occurring when $U_0 = 0.25$ m/s and $U_L = -0.25$ m/s. The largest error was when $U_0 = 0.1$ m/s and $U_L = -0.5$ m/s. Again, as for the ebb depths, the RPE showed the largest variation in response to changes in U_0 and U_L.

The same procedure was applied to the velocities from the table. Again when plotting each of the error measures for the ebb scenarios against U_0 and U_L, each error measure plot showed similar shaped patterns. The smallest errors occurred when $U_0 = U_L = 0.25$ m/s. The

worst results occurred for U_0 = 0.5 m/s and U_L = 0.25 m/s. Applying the same form of analysis to the velocities in the flood flow, apart from the RPE, the different error measures displayed identical patterns with the smallest error being when U_0 = 0.25 m/s, U_L = -0.25 m/s and the largest when U_0 = 0.5 m/s and U_L = -0.25 m/s. Again, as for the depths, the RPE showed the greatest sensitivity to changes in the values of U_0 and U_L.

5.6.2 River Medway

As for the Thames, the initial data was again generated over a 25 tidal cycle period and the time and spatial data remained the same. The latter [time and spatial data] thus implies the model will have 36 depth and 35 velocity sections. The zero Km-point upstream is near Rochester. Before a neural network simulation can be performed for different U_0 (at the Rochester end) and U_L at the 35 Km-point (at the confluence of the two rivers), it was necessary to check the matching of the depths and velocities of the generated data from the Medway with that of the Thames. The values of U_0 near Rochester would be just arbitrarily prescribed but necessarily smaller than the U_0 used for the Thames. It must be remembered of course that the generated velocities at the boundaries are not the same as the values of U_0 and U_L. As previously mentioned (cf. Chapter 3), U_0 is a constant (over a short period of time) freshwater input forcing velocity whilst U_L is the forcing velocity due to sinusoidal type gravitational tidal effects. In the cases that follow, as before, the generated depth and velocity values near the boundaries for both rivers were obtained by simple inspection of the data output.

Ebb tide: U_0 = 0.25, U_L = 0.5

This case corresponds to the Thames one as depicted by fig. 5.35. The generated depth and velocity in the case of the Thames at the 60 Km-location were 15.3154 m and 0.2486 m/s. However, generating data for the Medway starting at 0 Km (near Rochester) and finishing at the 35 Km-point, equivalent to the 60 Km-location in the Thames model, revealed differences in the two sets of data. The depth and velocity values generated at the estuarine end of the Medway were respectively, 15.3091 m and 0.2435 m/s. The depth was therefore down by 0.0063 m and the velocity by 0.0051 m/s.

The maximum recorded absolute difference between the neural network simulation and the target data in the depth was 0.0029 m whilst that for the velocity was 0.0009 m/s.

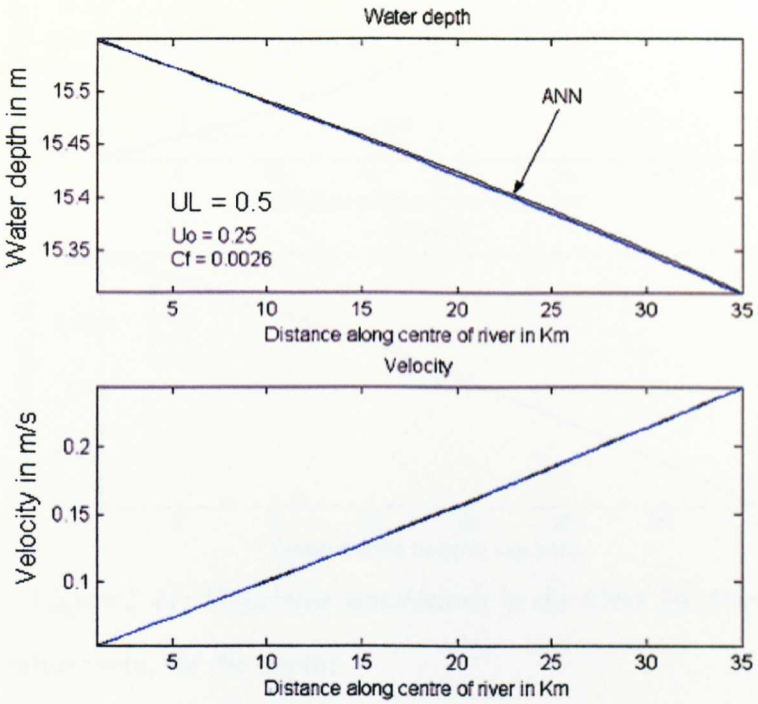

Figure 5.43: Ebb tide simulations in the River Medway

The error measure ratios were, first for the depth:

$$\frac{ME}{RMSE} \sim \pm 0.905, \quad \frac{ESD}{RMSE} \sim \pm 0.100 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.762$$

and then the velocity:

$$\frac{ME}{RMSE} \sim \pm 0.865, \quad \frac{ESD}{RMSE} \sim \pm 0.081 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.676$$

Flood tide: $U_0 = 0.25$, $U_L = -0.5$

This data generation and simulation corresponds to that depicted in fig. 5.36 (for the Thames). The generated depth and velocity for the Thames at the 60 Km-location were 15.1952 m and 0.1645 m/s. Likewise, those for the Medway at the 35 Km-point were 15.2525 m and 0.2109 m/s. Hence, the discrepancies between the generated data sets were 0.0573 m and 0.0464 m/s, approximately ten times larger than those for the ebb tide case. With regard to the neural network simulation, the maximum absolute difference in depth was 0.0031 m whilst that for the velocity was 0.0040 m/s.

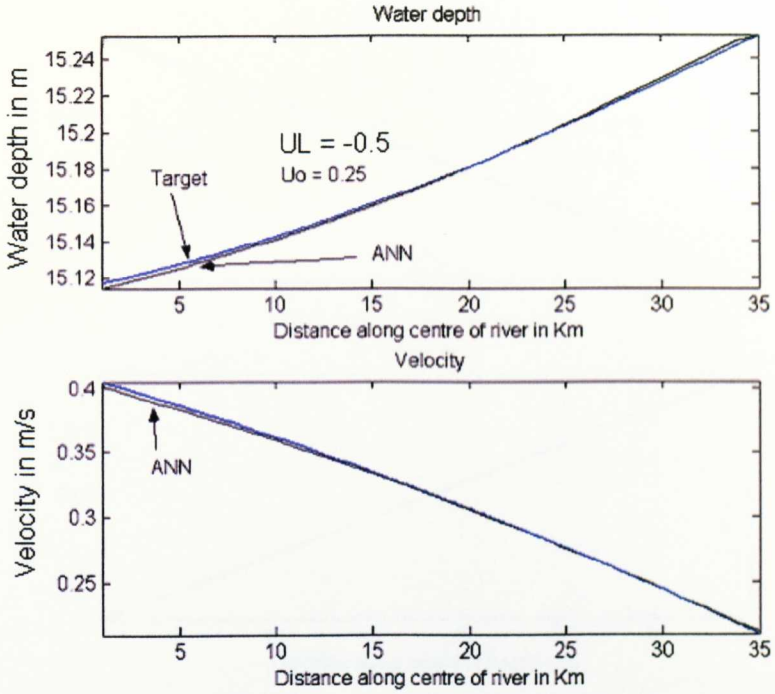

Figure 5.44: Flood tide simulations in the River Medway

The usual error ratios were, for the depth:

$$\frac{ME}{RMSE} \sim \pm 0.824, \quad \frac{ESD}{RMSE} \sim \pm 0.253 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.529$$

and the velocity:

$$\frac{ME}{RMSE} \sim \pm 0.737, \quad \frac{ESD}{RMSE} \sim \pm 0.279 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.947$$

Ebb tide: $U_0 = 0.15$, $U_L = 0.5$

The depth and velocity generated in the Thames at 60 Km (cf. fig. 5.35) were 15.3154 m and 0.2486 m/s whilst the corresponding values for the Medway at 35 Km were 15.1892 m and 0.1466 m/s. The discrepancies between the generated data are now 0.1262 m and 0.1020 m/s. These discrepancies are now significant being on average twenty times as large as the first case considered here for the Medway.

As for the neural network simulation, maximum absolute differences in depth and velocity were 0.0044 m and 0.0017 m/s.

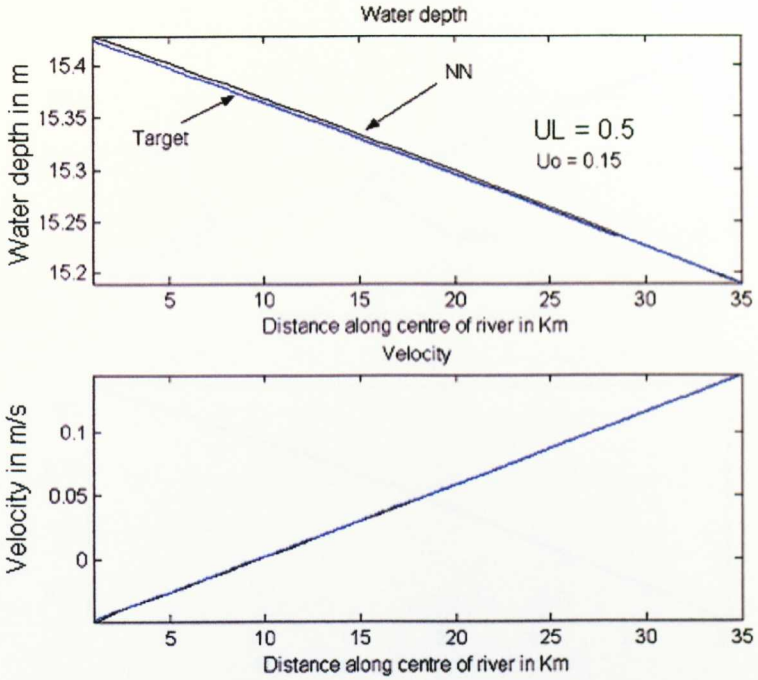

Figure 5.45: Ebb tide simulations in the River Medway

The depth error ratios were:

$$\frac{ME}{RMSE} \sim \pm 0.875, \quad \frac{ESD}{RMSE} \sim \pm 0.085 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.656$$

and similarly, for the velocity:

$$\frac{ME}{RMSE} \sim \pm 0.736, \quad \frac{ESD}{RMSE} \sim \pm 0.272 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.848$$

Flood tide: $U_0 = 0.15$, $U_L = -0.5$

Generated depth and velocity for the Thames at 60 Km (cf. fig 5.36) were 15.1952 m and 0.1645 m/s whilst those for the 35 Km-location in the Medway were 15.1469 m and 0.1254 m/s. The depth and velocity discrepancies were therefore 0.0483 m and 0.0391 m/s.

The discrepancies here are of the same order as those in the second case considered for the Medway. The maximum absolute differences in the neural network simulation for the depth and velocity were respectively, 0.0096 m and 0.0018 m/s.

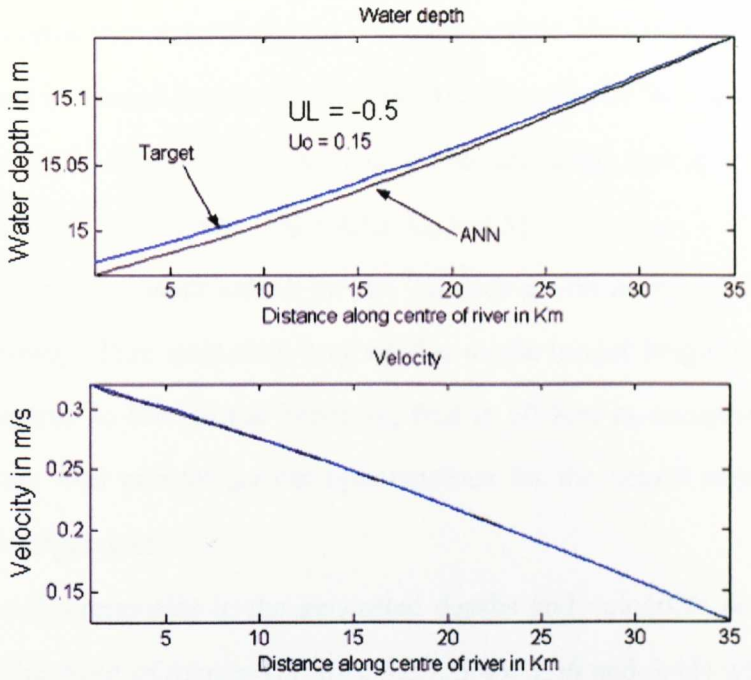

Figure 5.46: Flood tide simulations in the River Medway

The simulation resulted in the following error ratios, firstly for the depth:

$$\frac{ME}{RMSE} \sim \pm 0.885, \quad \frac{ESD}{RMSE} \sim \pm 0.311 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 5.934$$

and secondly for the velocity:

$$\frac{ME}{RMSE} \sim \pm 0.686, \quad \frac{ESD}{RMSE} \sim \pm 0.259 \quad \text{and} \quad \frac{RPE}{RMSE} \sim \pm 4.468$$

The following table summarises the error measures for the Medway simulations:

U_0 / U_L		RMSE	ME	ESD	RPE
0.25/0.5	depth	0.00210	0.0019	0.00021	0.01210
	velocity	0.00037	0.00032	0.00003	0.00210
0.25/-0.5	depth	0.00170	0.00140	0.00043	0.00940
	velocity	0.00190	0.00140	0.00053	0.00940
0.15/0.5	depth	0.00320	0.00280	0.00027	0.01810
	velocity	0.00045	0.00033	0.00012	0.00220
0.15/-0.5	depth	0.00610	0.00540	0.00190	0.03620
	velocity	0.00072	0.00049	0.00019	0.00320

Table 5.27: Comparison of simulation errors in the River Medway

The error measures for the depth, both flood and ebb, decreased in value as U_0 increased from 0.15 m/s to 0.25 m/s, the RPE showing the greatest relative decrease. As regards the

ebb velocity, the error measures displayed a similar pattern. However, for the flood velocity, the error measures increased in response to a positive increase in the value of U_0.

A comparison of tables 5.26 and 5.27, for compatible scenarios, that is

$\{U_0 = 0.15, U_L = -0.5\}$ and $\{U_0 = 0.15, U_L = 0.5\}$

revealed that the error measure values for the Thames are on average 2.5 times larger than those of the Medway. This inequality may be due to the longer length of simulation in the Thames as compared to that in the Medway, that is 60 Km as compared to 35 Km. The longer length may well present greater opportunities for the neural network simulation to deviate from the target data.

Returning to the discrepancies in the generated depths and velocities between the Thames and Medway at the point of merger (cf. figs. 5.35, 5.43, 5.36 and 5.44) where U_0 is 0.25 m/s in both cases, as previously noted, the differences were 0.0063 m and 0.0051 m/s for the ebb flow and 0.0573 m and 0.0464 m/s for the flood flow. These discrepancies are due to the difference in length of the two models, that is 35 Km for the Medway and 60 Km for the Thames. However, if the prescribed velocity U_0 for the Medway (near Rochester) is much less than that for the Thames, say 0.15 m/s, a not unreasonable assumption, then (cf. figs. 5.35, 5.45, 5.36 and 5.46) the discrepancies in the generated depth and velocities at the point of merger are now 0.1262 m and 0.1020 m/s for the ebb flow and 0.0483 m and 0.0391 m/s for the flood flow. This will undoubtedly require some modification to one of the finite difference equations to enable the matching of the depths and velocities as per a suggestion by Johns [50] or possibly using perturbation methods, Nayfeh [84] or asymptotic expansions, Rees [87]. This is worthy of further investigation (see Chapter 6). This would not have been a problem if real observed data had been available for both the Thames and Medway.

Notwithstanding the aforementioned comments, the neural network performed well on the simulation of the generated target data for the Medway.

5.7 Conclusion

A neural network model incorporating bottom friction, trained using the backpropagation algorithm with a bipolar sigmoid activation function, was first developed. The depth network was trained over 1200 Km in an effort to avoid overfitting (cf. Chapter 4) in an ebb tide regime, producing satisfactory results with a RMSE < 0.00084 and a maximum deviation between the target depth and the neural network simulation < 0.015 m. This maximum deviation occurred near the 0 Km-boundary. The velocity network was similarly trained producing satisfactory results once again with a RMSE < 0.0005 and a maximum deviation less than 0.0078 m/s near the 0 Km-boundary.

Following the same procedure as in Chapter 4, the validation of both networks was performed using the unseen flood tide data. The accuracy of the simulation of both networks decreased rapidly around the 1100 Km location producing errors of ~0.1 m and ~0.1 m/s and a RMSE ~ 0.046, as a consequence of the nonlinear bottom friction effects near the linearized boundaries. As a result, the architecture and parameters of the two networks were re-examined in an effort to develop a model that could cope more adequately with these errors at the 1100 Km location. After some degree of experimentation, it was decided to change the activation function to the LeCun version with appropriately modified weights. There was little significant improvement during training to the errors in the depth simulations, the deviation between target and simulation reducing from 0.15 m to 0.1 m but with some increase in the RMSE.

However, validation of the depth and velocity networks with the LeCun activation function produced significant improvements in the accuracies of the simulations. At approximately the 1100 Km location, the deviations in depth and velocity were reduced to less than 0.018 m and 0.007 m/s respectively. The deviations were then compared with the results of simulations using the original bipolar sigmoid activation function. The deviations from the latter had been reduced by a factor of five in the case of the depth network and a factor of fifteen for the velocity network. The finalized depth network is fully specified in fig. 4.1 and

the weights in table 5.17. Similarly, the finalized velocity network is fully specified by the weights in table 5.20 and fig. 4.1

To check the '*universal approximation*' properties of these two finalized neural networks, the models were tested using a number of different scenarios. Varying the depth between 10 m and 20 m indicated that the models coped well at 10 m. and to a lesser degree, at 20 m. There was much better agreement between 10 m and 15 m than between 15 and 20 m as expected. The models were also tested with different values of the coefficient of friction. Again the results were satisfactory except for a zero coefficient where errors of 0.05 m and 0.05 m/s were produced. The model was then tested using three different sets of values for U_L. The variation in the values had little effect on the errors, the maximum being 0.004 m for the depth tests.

Following this, simulations were then performed to see the effects of varying the value of the upstream velocity U_0. It was noted that a decrease in the value of U_0 from its training value of 0.25 produced more significant changes in the size of the errors than an increase beyond the training value. Even so, the maximum error in the depth was no larger than about 0.05 m. The increase in the errors in velocity were less significant. Finally, the model was tested by varying the values of U_0 and U_L at the same time. It was observed that the maximum error again occurred with the depth simulations, being of the order of 0.08 m, the velocity errors being smaller. It appeared that changing the value of U_0 had a much more significant effect on the accuracy of the simulation than changing U_L.

This chapter concluded with an application of the model to the area encompassing the confluence of the Thames and Medway rivers. The ebb tide simulations of the depth and velocity of the Thames, using the 1200 Km training values for U_0 and U_L, produced errors with maximum values of 0.0137 m and 0.004 m/s, the corresponding errors for the flood tide being 0.007 m and 0.0085 m/s. Increasing or decreasing the magnitude of U_L produced corresponding increases or decreases in the errors. Apart from the RPE, the other error measures displayed consistent behaviour in response to changes in the values of U_0 and U_L. The most accurate simulations were when U_0 was near to 0.25 m/s (the training value) as

expected but most inaccurate when U_0 approached the magnitude of the training value of U_L that is, 0.5 m/s. This is as it should be indicating that the freshwater input tends to uniquely determine the flow. Apart from the ESD, the error measures were between 1.15 and 1.35 times that of the 1200 Km simulations.

The simulations of the Medway produced maximum errors of 0.0029 m and 0.0009 m/s in the depth and velocity respectively when performed using the training boundary velocities of the ebb cycle. In flood tide mode with the same boundary values, the maximum errors were 0.0031 m and 0.0040 m/s. Using a smaller freshwater input that is, $U_0 = 0.15$, the errors in the ebb tide were 0.0044 m and 0.0017 m/s and correspondingly in the flood tide, 0.0096 m and 0.0018 m/s.

Comparing the neural network simulations of both the Thames and the Medway, it was noted that the depth and velocity error measure values for the Thames were approximately 2.5 times greater than the corresponding values for the Medway. Further, a comparison of the generated depth and velocity data at the precise confluence of the two rivers revealed discrepancies in the depths and velocities ranging from 0.0063 m to 0.1262 m and 0.0051 m/s to 0.1020 m/s. This may well necessitate some form of 'matching' between the finite difference models at this location.

Chapter 6
Discussion and Results

6.1 Discussion

Current research is focussing on developing a non-linear modelling methodology to represent river systems and in particular, the use of artificial neural networks (ANN). The latter have the advantage of being able to use field data directly without any simplification, unlike regression analysis wherein some assumption has to be made *a priori* as to the form of the equation. Most ANNs used in river modelling are of the multilayer type, trained using the backpropagation algorithm and it is this model architecture and training paradigm that is used in this research. Thus, the aim of this research was to develop a neural network model and apply it to river flows, in particular, the confluence of the rivers Thames and Medway.

Although the neural network modelling was reasonably successful, a main obstacle to the research work was the lack of available measured flow data in the area of interest. To overcome this, much effort was diverted into developing software to generate artificial data against which the models could be trained and validated. A main difficulty encountered when training neural networks is that they are prone to 'overfitting' if the amount of data is limited. However, as it was necessary to generate artificial data, as much data as was required could be created so that this 'difficulty' was totally negated. In fact, the arguments of Sarle [92] were used when generating this data to ensure overfitting was either limited or indeed eliminated.

Limiting the research to a narrower field, a one dimensional shallow water model was developed using finite differences based upon the St. Venant Equations wherein the advection terms are ignored. Stability, convergence and compatibility of the scheme were subsequently analysed mathematically. With no data available beyond the first time level, the leapfrog method was an obvious and sensible choice for the explicit finite difference scheme used in the research. The finite difference scheme model was then used to generate depth and flow data using three different depth scenarios in an ebb tide regime over a length of 600 Km using 25 tidal cycles. Without any imposed tidal velocity, the resulting depth

converged to the value as predicted by Laplacian theory. The results were in excellent agreement with that of a numerical model developed by Johns [51]. The finite difference scheme was then extended to include non-linear terms such as bottom friction wherein the variable density of the seawater was included. However, this proved to be difficult and so the density was set equal to a constant, the density of freshwater in this case. It was noted that, as suspected, the addition of bottom friction although converting the regular sinusoidal curve to an irregular one, it tended to 'dampen' out the curve and any possible errors in the finite difference scheme computation.

Having constructed a finite difference scheme model that could generate data with or without bottom friction, the research progressed towards the development of a neural network to simulate the depth and velocity of the river flow without bottom friction. Much experimental work ensued in order to obtain information on the optimum architecture of the hidden layer and the parameters such as the learning rate and momentum term. At this point, some concern arose over the problem of 'overfitting' when training the neural network over a length of 600 Km. Following the method of Sarle [92], 1080 data records would be required to ensure this was not a problem. However, this would have necessitated reducing the time step (30 s) and spatial distance (500 m) accordingly to maintain the Courant number within the correct limits to ensure the stability of the finite difference scheme. A simple analysis revealed that over 37000 time levels would be needed in the finite difference scheme but this required more computer memory than was available. The solution was relatively simple and solved by extending the length of simulation to 1200 Km and leaving the time and spatial values of the leapfrog grid as they were. This implied longer simulation times before a solution was reached.

With the depth network fully specified, the training simulations were performed with an error of less than 0.003 m and a RMSE of 0.0014, the major changes to the weights in the network being limited mostly to those between the input and hidden layers. Validation of the depth network was performed using flood tide (unseen) generated data. The maximum error was less than 0.008 m and the resultant RMSE and ME were respectively 0.0065 and

0.0043 that compared favourably with that of Dibike [31]. During the training of the velocity network, the maximum error was less than 0.016 m/s with a RMSE of 0.0011. The major changes to the weights were between the input and hidden layers that were about two to three times the changes between the other units. Validation of the network generated a very acceptable maximum error of 20 mm/s with RMSE and ME values of 0.0079 and 0.0067 that also compared favourably with the model of Dibike [31]. Finally, the two neural network simulations were compared with the numerical model of Johns [51] producing very good agreement.

The neural network developed initially was somewhat idealized in that it did not include the effects of bottom friction. Thus this model (a novel approach) was extended to include the effects of bottom friction so that it was much more representative of river flows. Although the one-dimensional shallow water equations have been widely used to represent river flows and in some cases, simulated using neural networks, the latter have not included bottom friction. The new models were trained (as for the zero friction models) using the backpropagation algorithm with a bipolar sigmoid activation function. The ebb tide depth training produced very satisfactory results such as a RMSE < 0.00084 and a maximum deviation < 0.015 m whilst the velocity training errors were a RMSE < 0.0005 and a maximum deviation less than 0.0078 m. In both cases concerning depth and velocity, these maximum errors occurred near the 0 Km-boundary.

Although very satisfactory results were obtained during the training phase, the validation was more difficult. The validation was performed as for the zero friction models using unseen flood tide generated data. However, the accuracy of the simulation of both networks decreased rapidly around the 1100 Km-location producing errors of ~0.1 m/s and a RMSE ~ 0.046. This was a consequence of the nonlinear bottom friction effects near the linearized boundaries. Both the architecture and parameters of the two networks were re-examined and found to be satisfactory. Attention was then focussed on the activation function and after a considerable degree of experimentation, a novel implementation of the LeCun version with appropriately modified weights was employed. There was little significant improvement

during training to the errors in simulations; however, there were significant improvements in the validation. At approximately the 1100 Km-location, the deviations in depth and velocity were reduced to less than 0.018 m and 0.007 m/s respectively. In fact, the deviations were reduced by a factor of five in the case of the depth network and a factor of fifteen for the velocity network.

The two finalized networks were then tested in a number of different scenarios to evaluate their '*universal approximation*' properties. The scenarios involved varying quantities such as the still water depth, the coefficient of friction C_D, the tidal forcing velocity U_L and the imposed freshwater flow U_0. The networks coped very well with a change of the still water depth to 10 m but not quite so well at 20 m. Testing the models with different values of the coefficient of friction were satisfactory except for $C_D = 0$ when errors of 0.05 m and 0.05 m/s were produced. The next set of tests involved variation of U_L, which displayed little change to the errors, the maximum error being 0.004 m for the depth tests. Varying the value of the upstream velocity U_0 had a much more significant effect on the accuracy of the simulations than varying U_L. Decreasing U_0 from its training value of 0.25 had a more significant effect on the depth errors than increasing it beyond the training value. Notwithstanding this, the maximum error in the depth was no larger than about 0.05 m. Changes to U_0 had less significant effect on the errors in the velocity. Varying U_0 and U_L at the same time, it was observed that the maximum error again occurred with the depth simulations being of the order of 0.08 m, the velocity errors being smaller.

Applying the neural network models to the River Thames (using now a length of only 60 Km) resulted in errors with maximum values of 0.0137 m and 0.004 m/s for the depth and velocity respectively in ebb tide mode, the corresponding errors for the flood tide being 0.007 m and 0.0085 m/s. Variation of U_L produced corresponding and consistent changes in the errors. The most accurate simulations were when U_0 was near to 0.25 m/s but most inaccurate when U_0 had a magnitude of 0.5 m/s, indicating for a fixed value of U_L, the freshwater input tends to uniquely determine the flow. The error measures were between 1.15 and 1.35 times that of the 1200 Km simulations. The simulations of the Medway (over

a 35 Km-length) resulted in maximum errors of 0.0029 m and 0.0009 m/s in the depth and velocity in ebb tide mode with U_0 = 0.25 m/s. In flood tide mode the resultant errors were 0.0031 m and 0.0040 m/s. Using a smaller freshwater input than that of the Thames, U_0 was changed to 0.15 m/s. The ebb tide errors increased by about a factor of 1.5 to 0.0044 m and 0.0017 m/s. The flood tide error in depth increased markedly to 0.0096 m whilst the velocity decreased to 0.0018 m/s.

Comparing the errors in the simulations of both rivers, the error measure values for the Thames were about 2.5 times the corresponding values for the Medway. Further, on comparing the generated (not simulated) depths and velocities at the precise confluence of the two rivers revealed discrepancies ranging from 0.0063 m to 0.1262 m and 0.0051 m/s to 0.1020 m/s. It should be noted that this is related to the finite difference modelling and not the neural network simulation. This may well necessitate some form of 'matching' between the generated data at this location as part of the finite difference scheme. These discrepancies apart, it was gratifying to observe the adaptability of the friction models with respect to variations in depth and coefficient of friction.

The simulations were performed on a 64-bit dual processor machine, the code being developed in a Matlab environment. Training with 1080 records in 12000 iterations took on average, 5 minutes, whilst the validation took less than a second.

Possible avenues of further research

- A useful direction would be to convert the friction models as developed in this research into a time series predictor of river flow depths and velocities. As it stands, the models will predict these quantities for every $1/16^{th}$. of a time cycle (cf. Chapter 3, fig. 3.3) but with a fixed freshwater input U_0, the cycle will be just repeated. Variable freshwater input in the form of a hydrographic time series would be required.
- The inclusion of variable density by varying the salinity. This would require a modification to equation **(3.25)** and could prove worthwhile. The friction model

coped reasonably well with variation in depth and friction and so it would be useful to assess its ability to varying density.

- Application of the neural network developed in this research to a three dimensional numerical model of the Thames such as that of Johns [52]. This model incorporates both variable depth and breadth as well as bottom friction. In addition, the nonlinear advection terms, equations **(3.1)** and **(3.2),** have been retained. This model generates higher tidal harmonics and so would be an interesting test of a neural network based on the research in this thesis.
- To investigate the slight disparity in the generated values from the finite difference scheme at the precise confluence of the rivers Thames and Medway. This might be possibly achieved using a matching technique of Johns [50] or by using asymptotic expansions, Rees [87]. This in no way affects the ability of the neural network model to simulate the rivers individually.
- The development of a support vector machine model would be an obvious continuation of this work. Structurally it is similar to a neural network so that the friction models could serve as a basis for its design. This could then be used for comparison with the neural network results and performance. To this end, a brief introduction to support vector machines has been provided in appendix D to aid in further research.

6.2 Results

The aim was to develop an artificial neural network (ANN) incorporating bottom friction to model flow parameters such as velocity and depth in the area of confluence of the rivers Thames and Medway.

A finite difference model based on the one dimensional shallow water equations was first developed to generate data to use for training and testing of the ANN. This was achieved with some considerable success. Comparison of the model with that of Johns [51] showed excellent agreement in both the velocity and depth generated data.

Following this, different ANNs with zero bottom friction were developed and tested in order to obtain information on the optimum network structure and related parameters. In particular, using guidelines suggested by Haykin [43], Marques de Sá [74], Hornik [45] et. al., after much simulation, a 3-7-1 MLP was developed. The model was trained using the backpropagation algorithm and demonstrated quite acceptable agreement with both the numerical model of Johns [51] and the neural network model of Dibike [31].

Using this zero bottom friction model as a template, the network was extended to include the effects of bottom friction. The model's activation function was replaced by a novel modified LeCun version to overcome conflicts near to the boundaries. After this change of activation, the network performed as it should near to the linearized boundaries.

The bottom friction neural network displayed good flexibility when confronted with a variation in the upstream and downstream imposed velocities. This 'universal approximation' capability was demonstrated even further when the still water depth or coefficient of bottom friction was altered.

Although there was a slight disparity in the generated (finite difference scheme) values for the depth and velocities at the precise confluence of the two rivers, individually, the network was able to simulate satisfactorily the river models over their different lengths.

References

1. Abrahart R and White S. Modelling sediment transfer in Malawi: Comparing backpropogation neural network solutions against a multiple linear regression benchmark using small data sets. *Phys. Chem. Earth (B)* 2001;26:19-24.
2. Abramowitz M and Stegun I A. *Handbook of Mathematical Functions*. New York: Dover Publications Inc; 1972.
3. Agrawal J and Deo M. Wave parameter estimation using neural networks. *Marine Structures* 2004:536-50.
4. Aitkenhead M, McDonald A, Dawson J, Couper G, Smart R, Billett M, Hope D and Palmer S. A novel method for training neural networks for time series prediction in environmental systems. *Ecological Modelling* 2003:87-95.
5. Anderson A and Rosenfeld E. *Neurocomputing: foundations of research*. Cambridge MA: MIT Press; 1988.
6. Asefa T, Kemblowski M, McKee M and Khalil A. Multi-time scale stream flow predictions: The support vector machines approach. *Hydrology* 2005;xx:1-10.
7. Atiya A F, El-Shoura S M, Shaheen S I and El-Sherif M S. A comparison between neural-network forecasting techniques - Case Study: river flow forecasting. *IEEE Transactions on Neural Networks* 1999;10:402-9.
8. Bertsekas D. *Nonlinear Programming*. Belmont, Massachussetts: Athenas Scientific; 1995.
9. Bishop C M. *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press; 1995.
10. Bose G and Jenkins G. *Time Series Analysis Forecasting and Control*. San Francisco: Holden Day; 1970.
11. Boser B, Guyon I and Vapnik V. A training algorithm for optimal margin classifiers. *Fifth annual workshop on computational learning theory*; 1992 p. 144-52.
12. Bowden G, Dandy G and Maier H. Input determination for neural network models in water resources applications. *Hydrology* 2004:75-92.

13. Bowden G, Dandy G and Maier H. Input determination for neural network models in water resources applications. *Hydrology* 2005:93-107.

14. Browne A. Neural Networks - personal communication.

15. Burges C. Simplified support vector decision rules. *Proc. 13th. International Conference of Machine Learning*; 1996 p. 71-7.

16. Burges C. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery* 1998;2: p. 121-67.

17. Callan R. *The Essence of Neural Networks*. London: Prentice Hall (Europe); 1999.

18. Carpenter G and Grossberg S. The ART of adaptive pattern recognition by a self organizing neural network. *Computer* 1988;21:77-90.

19. Chadwick A and Morfett J. *Hydraulics in Civil and Environmental Engineering*. London & New York: SPON Press; 1998.

20. Chang F and Chen Y. Estuary water-stage forecasting using radial basis function neural network. *Hydrology* 2003:158-66.

21. Cherkassky V and Mulier F. *Learning from Data: Concepts, theory and methods*. New York: John Wiley & Sons, Inc.; 1998.

22. Cigizoglu H K. Intermittent river flow forecasting by artificial neural networks. *DEVELOPMENTS IN WATER SCIENCE* 2002;47:1653-60.

23. Cigizoglu H K. Estimation, forecasting and extrapolation of river flows by artificial neural networks. *HYDROLOGICAL SCIENCES JOURNAL* 2003;48:349-62.

24. Cigizoglu H K. Estimation and forecasting of daily suspended sediment data by multi-layer perceptrons. *Advances in Water Resources* 2004:185-95.

25. Cortes C and Vapnik V. Support Vector Networks. *Machine Learning* 1995;20:273-97.

26. Delleur J, Tao P and Kavvas M. An evaluation of the practicality and complexity of some rainfall and runoff time series model. *Water Resources Research* 1976;12:953-70.

27. Dibike Y and Abbott M. Application Of Artificial Neural Networks To The Simulation Of A Two Dimensional Flow. *Hydraulic Research* 1999;37:435-46.

28. Dibike Y, Solomatine D and Abbott M. On the encapsulation of numerical-hydraulic models in artificial neural network. *Hydraulic Research* 1999;37.

29. Dibike Y, Minns A and Abbott M. Applications of artificial neural networks to the generation of wave equations from hydraulic data. *Hydraulic Research* 1999;37:81-97.

30. Dibike Y and Solmatine D. River Flow Forecasting Using Artificial Neural Networks. *Phys. Chem. Earth (B)* 2001;26:1-7.

31. Dibike Y. Developing generic hydrodynamic models using artificial neural networks. *Hydraulic Research* 2002;40:183-90.

32. Dibike Y and Coulibaly P. Temporal neural networks for downscaling climate variability and extremes. *Neural Networks* 2006;19:135-44.

33. Dolling O and Varas E. Artificial neural networks for streamflow prediction. *Hydraulic Research* 2002;40.

34. Drucker H, Burges C, Kaufman L, Smola A and Vapnik V. Support vector regression machines. *Advances in Neural Information Processing Systems*; 1997;9 p. 155-61.

35. El-RAbbany A, El-Diasty M and Raahemifar K. Report: *Sequential Tidal Height Prediction Using Artificial Neural Network.* Dept. of Civil Engineering, Ryerson University, Toronto 2003

36 Gerbeau J F and Perthame B. Derivation of viscous Saint-Venant system for laminar shallow water: Numerical validation. *Discrete and Continuous Dynamical Systems – Series B* 2001;1:89-102.

37. Girosi F. An equivalence between sparse approximations and support vector machines. *AI Memos* 1997.

38. Gurney K. *An Introduction to Neural Networks.* London et.al.: CRC Press; 1997.

39. Hammond M and Han D. Recession curve estimation for storm event separations. *Hydrology* 330, 3-4, 573-585, 2006.

40. Han D, Chan L and Zhu N. Flood forecasting using support vector machines. *Hydroinformatics* 2007:267-276.

41. Han D, Cluckie I D, Karbassioun D, Lawry J and Krauskopf B. River Flow Modelling Using Fuzzy Decision Trees. *Water Resources Research* 16, 6; 431-445, 2002.

42. Han D and Yang Z. River Flow Modelling Using Support Vector Machines. *Proceedings of the congress - international association for hydraulic research* 2001;CONF 29; VOL C:494-9.

43. Haykin S. *Neural Networks: a comphrensive foundation.* New Jersey: Prentice Hall International; 1999.

44. Hearst M, Dumais S, Suna E, Platt J and Scholkopf B. Support Vector Machines. *IEEE Intelligent Systems* 1998:18-28.

45. Hornik K. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks* 1989;2:359-66.

46. Huang W, Murray C, Kraus N and Rosati J. Development of a regional neural network for coastal water level predictions. *Ocean Engineering* 2003:2275-95.

47. Imrie C, Durucan S and Korre A. River flow prediction using artificial neural networks: generalisation beyond the calibration range. *Hydrology* 2000:138-53.

48. Izquierdo J, Perez R and Iglesias P. Mathematical models and methods in the water industry. *Mathematical and Computer Modelling* 2004:1353-74.

49. Jain A and Kumar A. Hybrid neural network models for hydrological time series forecasting. *Applied Soft Computing* 2006.

50. Johns B. Expert Lectures in coastal hydrodynamics - personal communication.

51. Johns B. One dimensional numerical river model - personal communication.

52. Johns B. Three dimensional numerical river model - personal communication.

53. Kachroo R. River flow forecasting. Part 1. A discussion of the principles. *Hydrology* 1992;133:1-15.

54. Kachroo R and Liang G. River flow forecasting. Part 2. Algebraic development of linear modelling techniques Volume 133, Issues 1-2, April 1992, Pages 1-15. *Hydrology* 1992;133:17-40.

55. Kachroo R, Sea C, Warsi M, Jemenez H and Saxena R. River flow forecasting. Part 3.Applications of linear techniques in modelling rainfall-runoff transformations,. *Hydrology* 1992;133:41-97.

56. Kachroo R, Liang G, Kang W and Yu X. River flow forecasting. Part 4. Applications of linear modelling techniques for flow routing on large catchments. *Hydrology* 1992;133:99-140.

57. Karunanithi N, Grenney W J, Whitley D and Bovee K. Neural Networks for River Flow Prediction. *JOURNAL OF COMPUTING IN CIVIL ENGINEERING* 1994;8:201-20.

58. Kerh T and Lee C. Neural networks forecasting of flood discharge at an unmeasured station using river upstream information. *Advances in Engineering Software* 2005:1-11.

59. Kisi O. River Flow Modelling Using Artificial Neural Networks. *Journal of hydrologic engineering* 2004;9:60-3.

60. Kocjancic R and Zupan J. Modelling of the river flowrate: the influence of the training set selection. *Chemometrics and Intelligent Laboratory Systems* 2000:21-34.

61. Kumar K D, Srinivasa Raju K and Sathish T. River Flow Forecasting using Recurrent Neural Networks. *Water resources management -dordrecht* 2004;18:143-61.

62. Lapedes A and Faber R. How Neural Nets Work. *Neural Information Processing Systems* 1988:442-56.

63. LeCun Y. Generalization and network design strategies; *Technical Report, University of Toronto* 1989.

64. LeCun Y. Efficient Learning and Second-Order Methods, a tutorial.. *Neural Information Processing Systems*; 1993.

65. LeCun Y, Bottou L, Orr G and Muller K. *Efficient BackProp*. Berlin: Springer-Verlag 1998.

66. Lin B and Falconer R. Numerical modelling of three-dimensional suspended sediment for estuarine and coastal waters. *Hydraulic Research* 1996;34:435-56.

67. Lippmann R. An introduction to computing with neural nets. *IEEE ASSP Magazine* 1987:4-22.

68. Maier H R and Dandy G C. Forecasting salinity using neural networks and time series models. *National Conference Publication – Institution of Engineers, Australia* 1994

69. Makarynskyy O. Improving wave predictions with artificial neural networks. *Ocean Engineering* 2004:709-24.

70. Makarynskyy O. Improving wave predictions with artificial neural networks. *Ocean Engineering* 2005:101-3.

71. Makarynskyy O, Pires-Silva A A, Makarynska D and Ventura-Soares C. Artificial neural networks in wave predictions at the west coast of Portugal. *Computers & Geosciences* 2005:415-24.

72. Mangasarian O. Linear and Nonlinear separation of patterns by linear programming. *Operations Research* 1965;13:444-52.

73. Mangasarian O. Multi-surface method of pattern separation. *IEEE Transactions on Information Theory* 1968;IT 14:801-7.

74. Marques de Sá J.P. *Pattern Recognition. Concepts, Methods and Applications.* Berlin, Heidelberg, New York: Springer-Verlag; 2001.

75. Matalas N. Mathematical assessment of symmetrical hydrology. *Water Resources Press* 1967;3:937-45.

76. Mattera D and Haykin S. *Support vector machines for dynamic reconstruction of a chaotic system.* Cambridge MA: MIT Press; 1999.

77. McCulloch W S and Pitts W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology* 1943; 5:115-133

78. Mercer B. Functions of positive and negative type, and their connection with the theory of integral equations. *Transactions of the London Philosophical Society* 1909;209:415-46.

79. Minsky M and Papert S. *Perceptrons.* Cambridge, MA: MIT Press; 1969.

80. Moradkhani H, Hsu K, Gupta H and Sorooshian S. Improved streamflow forecasting using self-organizing radial basis function artificial neural networks. *Hydrology* 2004:246-62.

81. Mujumdar P P. Flood Wave Propagation - The Saint Venant Equations. *Resonance, Indian Academy of Sciences* 2001:6(5): 66-73.

82. Muller K, Smola A, Ratsch G, Scholkopf B, Kohlmorgen J and Vapnik V. Predicting Time Series with Support Vector Machines. *ICANN'97*; 1997;1327 p. 999-1004.

83. Nayak P, Sudheer K, Rangan D and Ramasastri K. A neuro-fuzzy computing technique for modelling hydrological time series. *Hydrology* 2004:52-66.

84. Nayfeh A H. *Perturbation Methods*. Wiley, J. 1973

85. Picton P. *Neural Networks*. Basingstoke: Macmillan; 2000.

86. Ponce V M and Simons D B. Shallow wave propagation in open channel flow. *Journal of the Hydraulics Division, ASCE* 1977;103:1461-1476.

87. Rees L H. *An asymptotic expansion approach to a model simulating dissipative waves on a plane beach.* MPhil thesis. London Guildhall University 1994

88. Riad S, Mania J, Bouchaou L and Najjar Y. Artificial neural networks models for river flow prediction. *TRIBUNE DE L EAU* 2004;57:19-24.

89. Ripley B D. *Pattern Recognition and Neural Networks*. Cambridge: Cambridge University Press; 1996.

90. Rosenblatt F. The Perceptron:A probabilistic model for information storage and organization in the brain. *Psychological Review* 1958;65:386-408.

91. Sandhu N, Finch R and Chang F. Modelling flow-salinity relationships in the Sacramento-San Joaquin Delta using artificial neural networks. Technical Information Record OSP-99-1 California Dept. of Water Resources, Sacramento. C.A. 1999.

92. Sarle W. Neural Network FAQ; Available from URL:
ftp://ftp.sas.com/pub/neural/FAQ.html
or

http://vision.eng.shu.ac.uk/neural/FAQ/FAQ.html

93. Schleider O and Cser J. Joint Workshop on Neural Networks in Civil Engineering April 1999, Delft; Available from URL: http://pc1.bauinf.tu-cottbus.de/veranstaltungen/Neural99

94. Scholkopf B, Burges C and Smola A. *Advances in Kernel Methods - Support Vector Learning*. Cambridge, Massachussetts: MIT Press; 1998.

95. Sivakumar B, Jayawardena A and Fernando T. River flow forecasting: use of phase-space reconstruction and artificial neural network approaches. *Hydrology* 2002:225-45.

96. Skapura D. *Building Neural Networks*. New York: ACM Press; 1996.

97. Smith G. *Numerical Solution of Partial Differential Equations*. London: OUP; 1974.

98. Sonnenborg T O. Groundwater Research Centre Annual Report. Dept. of Hydrodynamics and Water Resources, Technical University of Denmark, 1998

99. Tarassenko L. *A Guide to Neural Computing Applications*. London: Arnold; 1998.

100. Tawfik M. Linearity versus non-linearity in forecasting Nile river flows. *Advances in Engineering Software* 2003;34:515-24.

101. Thirumalaiah K and Deo M C. River Stage Forecasting Using Artificial Neural Networks. *JOURNAL OF HYDROLOGIC ENGINEERING* 1998;3:26-32.

102. Turban E and Aronson J E. *Decision Support Systems and Intelligent System:* New Jersey:Prentice-Hall; 2001.

103. Vapnik V and Lerner A. Pattern recognition using generalized portrait method. *Automation and Remote Control* 1963;24.

104. Vapnik V and Chervenenkis A. *On the uniform convergence of relative frequencies of events to their probabilities*; 1971.

105. Vapnik V. *Statistical Learning Theory*. New York: Wiley & Sons;; 1998.

106. Vreugdenhil C. *Computational Hydraulics*. New York: Springer-Verlag; 1989.

107. Werbos P. *Beyond regression: New tools for prediction and analysis in the behavioural sciences*. Cambridge MA: Harvard; 1974.

108. Wheeler A. *The Tidal Thames*. Oxon Routledge & Kegan. 1979

109. Widrow B and Hoff M. Adaptive switching circuits. *IRE WESCON Convention record* 1960:96-104.

110. Yang Z and Han D. Derivation of unit hydrograph using a transfer function approach. *Water Resources Research* 42; 2006.

111. Zealand C, Burn D and Simonovic S. Short term streamflow forecasting using artificial neural networks. *Hydrology* 1999:32-48.

Appendix A – nomenclature

U_O	Freshwater (upsteam) forcing velocity
U_L	Tidal (downstream) imposed forcing velocity
h	still water level depth
ξ	water level above/below still water level
ρ	density of water
α	Momentum term
η	Learning rate
c	Wave celerity
T_P	Tidal cycle
g	Gravitational acceleration
C_D	Coefficient of friction
Cr	Courant number
u	Depth averaged velocity
x, t	Distance, time
T	Truncation error
i,j	subscripts
$O(.)$	Order
ANN	Artifical neural network
MLP	Multilayer perceptron
SVM	Support vector machines
ESD	Error standard deviation
ME	Mean absolute error
RMSE	Root mean squared error
RPE	Relative percentage error
MISO	Multiple input single output
SISO	Single input single output
SOM	Self-Organizing Map

Appendix B

Finite Difference Formulae

Notation for a function of one variable

Given that a function f and its derivatives are single valued, finite and continuous functions of x, then by Taylor's theorem:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + O(h^4) + \ldots \quad \textbf{(B1)}$$

and

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + O(h^4) + \ldots \quad \textbf{(B2)}$$

where the notation $f'(x)$, $f''(x)$ represent the derivatives $\frac{df}{dx}$, $\frac{d^2f}{dx^2}$ etc. and h is a small change in the value of x. Note that many authors use the notation Δx and not h.

Addition and subtraction of equations **(B1)** and **(B2)** result in respectively:

$$f(x+h) + f(x-h) = 2f(x) + 2\frac{h^2}{2!}f''(x) + O(h^4) + \ldots \quad \textbf{(B3)}$$

$$f(x+h) - f(x-h) = 2hf'(x) + 2\frac{h^3}{3!}f'''(x) + O(h^5) + \ldots \quad \textbf{(B4)}$$

Equation **(B3)** gives after some rearrangement

$$f''(x) \simeq \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \quad \textbf{(B5)}$$

ignoring terms of the $O(h^4)$ so that there is a leading error term of $O(h^2)$, whilst equation **(B4)** gives

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h} \quad \textbf{(B6)}$$

ignoring terms of the $O(h^5)$ where again there is a leading error term of $O(h^2)$.

Equation **(B6)** is referred to as a central difference approximation. From equation **(B1)** can be deduced the following forward difference approximation:

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h} \quad \textbf{(B7)}$$

and from equation **(B2)**, the backward difference approximation:

$$f'(x) \simeq \frac{f(x)-f(x-h)}{h} \qquad \textbf{(B8)}$$

Notation for a function of two variables

If f is now a function of two variables, say x (for distance) and t (for time) and divide the x–t plane into sets of equal rectangles (a mesh) such that $\delta x = h$ and $\delta t = k$, then the coordinates (x,t) for a mesh point P are

$$x = ih \text{ and } t = jk \qquad \forall j,k \in \mathbb{Z}^+$$

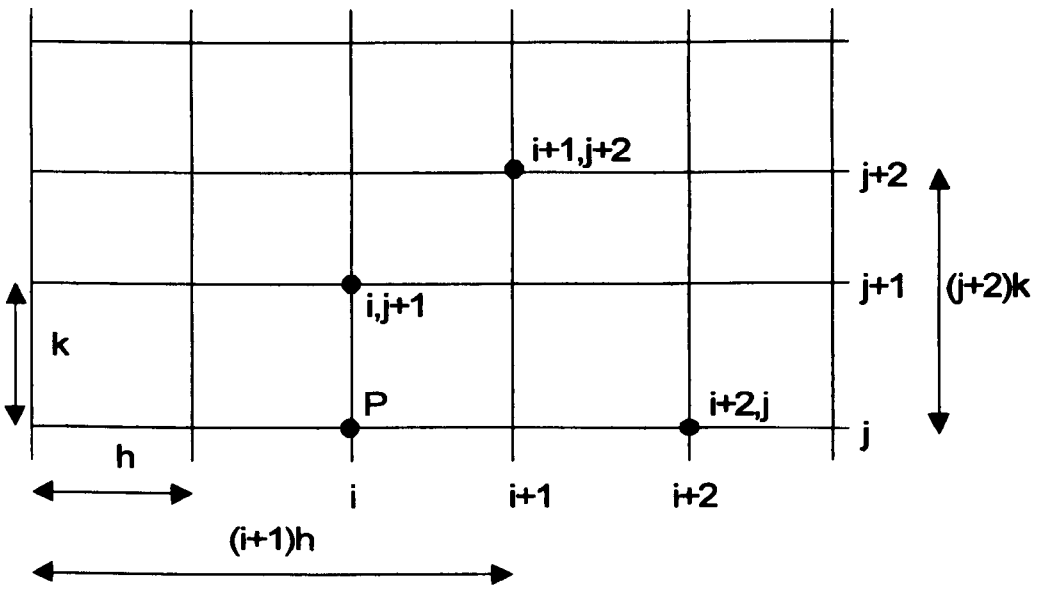

Figure B1: Leapfrog mesh

With this notation, then from equation **(B5)** if f is a function of both x and t:

$$\left(\frac{\partial^2 f}{\partial x^2}\right)_{i,j} \equiv f''(x)_{i,j} \simeq \frac{f[(i+1)h, jk]-2f[ih, jk]+f[(i-1)h, jk]}{h^2} \qquad \textbf{(B9)}$$

which is usually abbreviated to

$$\left(\frac{\partial^2 f}{\partial x^2}\right)_{i,j} \simeq \frac{f_{i+1}^j - 2f_i^j + f_{i-1}^j}{h^2} \qquad \textbf{(B10)}$$

As before, equation **(B10)** has a leading error term of $O(h^2)$. Note that some authors use a more cumbersome annotation. They do not use superscripts and so would have written equation **(B10)** thus:

$$\left(\frac{\partial^2 f}{\partial x^2}\right)_{i,j} \simeq \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{h^2}$$

Using similar arguments, equation **(B5)** gives in terms of the other independent variable t

$$\left(\frac{\partial^2 f}{\partial t^2}\right)_{i,j} \simeq \frac{f_i^{j+1} - 2f_i^j + f_i^{j-1}}{k^2} \qquad \textbf{(B11)}$$

with a leading error term of $O(k^2)$.

Using the same notation for x and t, the *central difference approximations* are:

$$\left(\frac{\partial f}{\partial x}\right)_{i,j} \simeq \frac{f_{i+1}^j - f_{i-1}^j}{2h} \qquad \textbf{(B12)}$$

$$\left(\frac{\partial f}{\partial t}\right)_{i,j} \simeq \frac{f_i^{j+1} - f_i^{j-1}}{2k} \qquad \textbf{(B13)}$$

the forward difference approximations:

$$\left(\frac{\partial f}{\partial x}\right)_{i,j} \simeq \frac{f_{i+1}^j - f_i^j}{h} \qquad \textbf{(B14)}$$

$$\left(\frac{\partial f}{\partial t}\right)_{i,j} \simeq \frac{f_i^{j+1} - f_i^j}{k} \qquad \textbf{(B15)}$$

and the backward difference approximations:

$$\left(\frac{\partial f}{\partial x}\right)_{i,j} \simeq \frac{f_i^j - f_{i-1}^j}{h} \qquad \textbf{(B16)}$$

$$\left(\frac{\partial f}{\partial t}\right)_{i,j} \simeq \frac{f_i^j - f_i^{j-1}}{k} \qquad \textbf{(B17)}$$

Appendix C:
Matlab Computer Code

The following programs are listed in this appendix. All Matlab comments are shown in italics for clarity. All tokenized words are in bold. Each program contains several lines of comments at the beginning indicating the purpose of each program and so will not be repeated here.

1. **Filename: VenantDataGenerator2.m**

   Creates both depth and velocity data for training and validation of the neural networks.

2. **Filename: DepthSolution.m**

   Used to train the depth neural network.

3. **Filename: VelocitySolution.m**

   Used to train the velocity neural network

4. **Filename: CombinedNetwork.m**

   This file replaces the two program files DepthSolutionValidation and VelocitySolutionValidation. This speeds up the validation and reduces the workload on the hardrive.

5. **Filename: DepthWeightFileCreation.m**

   Used to manually create neural network weights where required. There is also a companion file VelocityWeightFileCreation that is not listed as the code is almost identical. A note to that effect is included at the end of this program listing.

6. **Filename: RMSTest.m**

   Used to check and plot the RMS calculations.

```
% Filename: VenantDataGenerator2.m
% L.H.Rees Feb 2007
% Code to generate test data for Saint Venant Equations. Amended to convert the
% velocities to depth averaged ones i.e. in these equations, change u to u/(H(i,j)+ho)
% etc. The results agree perfectly with those of B.Johns.
% This is a 1D model.Initial condition is at t = 0 i.e. n = 1. Boundary conditions are at
% x = 0 i.e. j = 1 and x =(sections-1)/2 Km i.e. j = sections. Hence initial conditions
% are that velocity = 0 for all x and t = 0 except at x = 0 where we must have some
% basic input to start the system going. This is equivalent to a hydrographic input
% (boundary condition) at the river end of the estuary. This program computes using a
% FDS, the height above still water level and associated velocities, at the centre of the
% river given this boundary hydrographic velocity input at the river end of the estuary
% and a value for the average depth (still water level) of the channel.
% As we are using a staggered grid, the spatial interval is 2 x deltax. Velocities are
% stored in U(j,n) and the height values are stored in H(j,n). Using two separate
% matrices like this means that each matrix will have alternate rows containing
% nothing other than zeroes (even after computing) as a consequence of the finite
% difference scheme staggered method. The j index represents the rows of velocity and
% height respectively whilst the n index represents the time iteration levels.
% The velocity and depth estimates (sections) are obtained using a given number of
% time iterations. The number of tidal cycles (for time iteration) and updstream and
% downstream velocities are user selectable. This model covers a length of
% (sections-1)/2 km along the river. Note, for stability of the FDS scheme, the time and
% spatial grid has been chosen so that the Courant number is less than one.
% Suitable boundary conditions as suggested by B.Johns are used to avoid wave
% reflection. NOTE: AMPLITUDE = UL*SQRT(H/G) at the downstream boundary
% in B.Johns Fotran model.
% NOTE: on various lines of the code, a term called a divisor is used to calculate the
% depth averaged velocities (by dividing the velocities by it) where divisor represents
% the depth of the water at or near to the point of interest.
% With h = 15m and g = 9.81 then wavelength = tidal period x phase speed = (12.4 x
% 3600) x sqrt(g x h) = 541.5 Km approx.
% This program will generate data with or without bottom friction.
%-----------------------------------------------------------------------------------------------
% GET INITIAL DATA
%------------------------
deltax=500;                    % 1 Km x 50 sufficient for the Thames/Medwa
                                  area
deltat=30;                     % 30 seconds time interval
Tp = 12.4 * 3600;              % one tidal cycle converted to seconds
disp('Wave celerity = sqrt(gravity x still water depth).')
disp('So for still water depth = 20m, celerity = 14 m/s, for depth = 15m, celerity = 12.13
m/s, depth = 10m, celerity = 9.9m/s')
disp('As the number of required tidal cycles needs to be iterated through about ten times,
then as a guide:')
disp('no. of cycles for 50km = 10, no. of cycles for 100km = 20, no. of cycles for 200 km =
40, no. of cycles for 600 km = 120 etc.')
disp(' ')
```

```
tidalcycles = input ('How many tidal cycles do you wish to iterate over (note 12.4 hours =
one tidal cycle) ? ')
timesteps = Tp/deltat;                  % calculate the no. of timesteps required per tidal
                                        % cycle to fit the oscillatory wave
                                        % i.e. so that the sine wave oscillation changes sign
                                        % at 6.2 hours to reflect a tidal change
extratimesteps = (tidalcycles-fix(tidalcycles))*timesteps;
                                        % calculates extra timesteps needed if more than an
                                        % exact no. of tidal cycles.
sections=input('Enter no. of sections required (must be odd). e.g. 101 spans 50 Km, 201
spans 100 Km etc.) ? ');
H=zeros(sections, timesteps+extratimesteps);
U=zeros(sections-1,timesteps+extratimesteps);
ho = input ('Enter a value for the still water depth  ');
                                        % assumed initial condition at t = 0
velocity1 = input ('Enter a value for the upstream velocity ');
                                        % assumed boundary condition at t = 0, x = 0
velocity2 = input ('Enter a value for the downstream velocity ');
                                        % assumed boundary condition at t = 0, x = 50km
g=9.81;                                 % gravitational acceleration
c=sqrt(g*ho);                           % wave celerity: shallow water waves approximation
Cr=c*deltat/deltax;                     % Courant no. which must be <= 1 for stability
k=Cr/2*sqrt(ho/g);
m=-k;
l=1;
p=Cr/2*sqrt(g/ho);
r=-p;
q=l;
disp('Enter a value for the friction coefficient, say zero for no bottom stress or for example
0.0026 ');
Cf = input ('Value ? ');
bottomstress = 0;
% CALCULATE THE DATA FOR THE DEPTHS AND VELOCITIES FOR MATRIX H
% AND U FOR A WHOLE NUMBER OF TIDAL CYCLES
% ----------------------------------------------------------------------------------------
z1 = fix(tidalcycles);
for z=1:z1;
        for n =1:timesteps-1;
                divisor = H(3,n)+ho;    % calculating total depth at/or close to the
                                        % upstream boundary
                H(1,n+1)=sqrt(ho/g)*(2*velocity1-U(2,n)/divisor);    % using depth
                                        % average velocity i.e. U()/divisor
                                        % Bryan John's upstream boundary condition
                divisor = H(sections-2,n)+ho;    % calculating total depth at/or close to
                                                 % downstream boundary
                H(sections,n+1)=sqrt(ho/g)*(2*velocity2*sin(2*pi*(n-
                        1)*deltat/Tp)+U(sections-1,n)/divisor);
                                        % using depth averaged velocity - Bryan John's
                                        % downstream boundary condition
```

```
            for j = 3:2:sections-2;
                divisor = H(j,n)+ho;
                H(j,n+1)= l*H(j,n)+(k*U(j-1,n)+m*U(j+1,n))/divisor;
                                % stores depths for all the interior points
            end;
            for j=1:2:sections-2;
                divisor = H(j+2,n+1)+ho;
                if n > 1
                    bottomstress = 1 + Cf*abs(U(j+1,n-
                1))*deltat/(divisor*divisor);
                                % calculate the bottom stress (density = 1)
                                % note double division here as it is u * abs(u)
                else
                    bottomstress = 1;
                end
                U(j+1,n+1)=q*U(j+1,n)+(p*H(j,n+1)+r*H(j+2,n+1))*divisor;
                                % stores the velocities for the interior points
                U(j+1,n+1)=U(j+1,n+1)/bottomstress;
            end;
        end;
        if z <= z1                  % check to see if no. of required WHOLE tidal
                                    % cycles has been surpassed
            j=[1:2:sections];
                H(j,1)=H(j,n+1);    % if not, transfers latest iteration from
                                    % previous cycle to next one
            j=[2:2:sections-1];
                U(j,1)=U(j,n+1);    % if not, transfers latest iteration from
                                    % previous cycle to next one
        end;
end;
% CALCULATE THE DATA FOR THE DEPTHS AND VELOCITIES FOR MATRIX H
AND U FOR THE PORTION OF A TIDAL CYCLE
% ------------------------------------------------------------------------------------------
z2 = tidalcycles - z1;
if z2~=0
        for n =timesteps-1:timesteps+extratimesteps-1;
            divisor = H(3,n)+ho;        % calculating total depth at/or close to the
                                        % upstream boundary
            H(1,n+1)=sqrt(ho/g)*(2*velocity1-U(2,n)/divisor); % using depth averaged
                                % velocity i.e. U()/divisor - Bryan John's upstream
                                % boundary condition
            divisor = H(sections-2,n)+ho;       % calculating total depth at/or close to
                                                % downstream boundary
            H(sections,n+1)=sqrt(ho/g)*(2*velocity2*sin(2*pi*(n-
                1)*deltat/Tp)+U(sections-1,n)/divisor);     % using depth averaged
                                % velocity – John's downstream boundary condition
            for j = 3:2:sections-2;
                divisor = H(j,n)+ho;
```

```
                H(j,n+1)= l*H(j,n)+(k*U(j-1,n)+m*U(j+1,n))/divisor;
                                        % stores depths for all the interior points
            end;
            for j=1:2:sections-2;
                divisor = H(j+2,n+1)+ho;
                if n > 1
                    bottomstress = 1 + Cf*abs(U(j+1,n-
                1))*deltat/(divisor*divisor);
                                        % calculate the bottom stress (density = 1)
                else                    % double division here as it is u * abs(u)
                    bottomstress = 1;
                end
                U(j+1,n+1)=q*U(j+1,n)+(p*H(j,n+1)+r*H(j+2,n+1))*divisor;
                                        % stores the velocities for the interior points
                U(j+1,n+1)=U(j+1,n+1)/bottomstress;
            end;
        end;
end;
% CONVERSION OF VELOCITIES TO DEPTH AVERAGED ONES
%-------------------------------------------------------------------------
for j = 1:2:sections-2;             % having completed the iteration of the tidal cycles,
        divisor = H(j,timesteps+extratimesteps)+ho; % a final total depth is calculated
                                        % for each section and then
        U(j+1,n+1)=U(j+1,n+1)/divisor;  % all the final iterated velocities
                                        %  are converted to depth averaged ones.
        U(j+1,n)=U(j+1,n)/divisor;      % also all the penultimate final iterated
                                        % velocities are converted to depth averaged
                                        % ones - needed for the ANN.
end;
% PLOT THE DEPTH GRAPH
% -------------------------------
averagedepth=0;
for j=1:2:sections;
        averagedepth=averagedepth+H(j,timesteps+extratimesteps);
                                        % sum up all the calculated depths
end
averagedepth = averagedepth/((sections+1)/2)+ho; % obtain an average depth
                                        %  throughout the length of the river
j=[1:2:sections];
        LatestDepth=[H(j,timesteps+extratimesteps)];    % hold depths for plotting
        PreviousDepth=[H(j,timesteps+extratimesteps-1)];
        distance = round((j/2)*2*deltax/1000)-1;
                                        % calculate the distances for plotting
        subplot(2,1,1), plot (distance,LatestDepth+ho,'k-')
                        % plot of water depth along centre of river at different distances
        hold on;
        subplot(2,1,1),line(round(j/2)-1,averagedepth);
                        % plot the average depth throughout the length of the river
axis ([-inf, inf, -inf, inf])
```

```
xlabel ('Distance along centre of river in Km')
ylabel ('Water depth in metres')
title (['Water depth after ', num2str(tidalcycles), ' tidal cycles'])
% PLOT THE VELOCITY GRAPH
% ----------------------------------
averagevelocity=0;
for j=2:2:sections-1;
        averagevelocity=averagevelocity+U(j,timesteps+extratimesteps);
                                                    % sum up all the  calculated velocities
end
averagevelocity = averagevelocity/((sections-1)/2);
                        % obtain an average velocity throughout the length of the river
j =[2:2:sections-1];
        LatestSpeed=[U(j,timesteps+extratimesteps)];        % hold velocities for plotting
        PreviousSpeed=[U(j,timesteps+extratimesteps-1)];
        distance = round((j/2)*2*deltax/1000)-1;
                                                    % calculate distances for plotting
        subplot(2,1,2), plot (distance,LatestSpeed,'k-')
                        % plot velocity along centre of river at different distances
        hold on;
        subplot(2,1,2),line(round(j/2)-1,averagevelocity);
                        % plot average velocity throughout the length of the river
axis([-inf inf -inf inf])
xlabel ('Distance along centre of river in Km')
ylabel ('Velocity in metres per sec')
title (['Velocity after ', num2str(tidalcycles), ' tidal cycles'])
% SAVE DEPTH AND VELOCITY DATA TO A FILE
% ----------------------------------------------------------
depthsize = size(LatestDepth);
speedsize = size(LatestSpeed);
save VenantOriginal ho speedsize depthsize LatestSpeed PreviousSpeed LatestDepth
  PreviousDepth;                                    % save data for later use with the NN
```

```
% Filename DepthSolution.m
% Filename DepthSolution.m
% L.H.Rees March 2007
% USE THIS PROGRAM ALSO FOR VALIDATING DIFFERENT NEURAL
% NETWORK ARCHITECTURES (DIFFERENT NOS. OF HIDDEN NEURONS).
% This is a single hidden layer MLP using the sigmoid or tanh activation
% functions using the backpropogation algorithm, learning rate and momentum
% terms.Designed for file input of data and incremental updating
% THIS PROGRAM SOLVES FOR THE HEIGHTS ABOVE THE STEADY STATE
% WATER LEVEL. SEE VelocitySolution.m FOR THE VERSION THAT SOLVES
% THE VELOCITIES.
% THIS PROGRAM USES THE DATA FILE VenantOriginal to acquire the original
% data created by VenantGenerator2.
% GET INITIAL DATA ETC.
% ----------
prompt1 = 'This NN is designed to use three possible types of activation functions';
prompt2 = '1 - The sigmoid (logistic) function';
prompt3 = '2 - The tanh function';
prompt4 = '3 - The LeCun tanh function';
disp(prompt1), disp(prompt2), disp(prompt3), disp(prompt4)
disp(' ')
activation = input('Your choice for the activation function i.e. 1, 2 or 3 from above ? ');
disp(' ')
eta = input('Choice for learning rate e.g. usually 0.1 to 1 ? ');
disp(' ')
alpha = input('Value for the momentum term e.g. usually 0.1 to 0.8 ? ');
disp(' ')
decay1 = input('Value for weight decay rate from the input to hidden layer e.g. 0 to 0.005 ?
');
disp(' ')
decay2 = input('Value for weight decay rate from the bias to hidden layer e.g. 0 to 0.005 ? ');
disp(' ')
decay3 = input('Value for weight decay rate from the hidden to output layer e.g. 0 to 0.005 ?
');
disp(' ')
decay4 = input('Value for weight decay rate from the bias to output layer e.g. 0 to 0.005 ? ');
disp(' ')
requiredrms = input('Required value for the root mean square error e.g. 0.01 ? ');
disp(' ')
L1 = input('No. of neurons in the hidden layer e.g. 3 ? ');
disp(' ')
epochs = input('How many epochs i.e. iterations, do you require ? ');
disp(' ')
load VenantOriginal ho speedsize depthsize LatestSpeed PreviousSpeed LatestDepth
PreviousDepth;
disp('The pattern of final and penultimate veloocities and depths has now been loaded')
disp(' ')
N1 = 3;                                % No. of neurons in the input layer
```

```
P1 = 1;                                   % No. of neurons in the output layer
factor = input('Enlargement factor for rms plot e.g. 10 to 100 ? ');
                        % multiplier to enlarge rms plots since some might not show
% SETUP AND DIMENSIONALIZE ARRAYS
% ------------------------------------------------
BiasHidden = zeros(L1,1);     % Contains the weights of the bias units for the hidden layer.
Value of bias is +1
BiasOut = zeros(P1,1);        % Contains the weights of the bias units for the output layer.
Value of bias is +1
Patterns = zeros(depthsize(1,1)-2,N1);      % Input data matrix holding all the patterns
Win = zeros(N1,L1);                   % Weights for the input to hidden layers matrix
                                      % Each row separately, must represent all of
                                      % the weights for a given input unit
Wout = zeros(L1,P1);                  % Weights for the hidden to output layers matrix
                                      % Each row separately, must represent all of the
                                      % weights for a given hidden unit
Tg = zeros(depthsize(1,1),P1);              % Target data matrix.
NetIn = zeros(L1,depthsize(1,1)-2);   % Matrix to present summation results to the
                                      % hidden layer
Oh = zeros(N1,L1);                    % Same dimensions as NetIn1. Used to store
                                      % activated values of NetIn1
WedOut = zeros(L1,P1);                % Weight error derivatives for output to hidden
                                      % layers.
WedIn = zeros(N1,L1);                 % Weight error derivatives for hidden to input layers.
OldWedOut = zeros(L1,P1);             % Weight error derivatives for output to hidden layers
                                      % from previous calculation for use with momentum
                                      % terms.
OldWedIn = zeros(N1,L1);              % Weight error derivatives for hidden to input layers
                                      % from previous calculation for use with momentum
                                      % terms
ErrorHidden = zeros(L1,1);            % error signals for hidden to input weight adjustments
DeltaHidden = zeros(L1,1);            % delta adjustment term for hidden to input weight
                                      % modifications
OldDeltaHidden = zeros(L1,1);         % Contains the old delta values to use with the hidden
                                      % bias units and the momentum term.
ErrorOut = zeros(P1,1);               % error signals for output to hidden weight
                                      %  adjustments
DeltaOut = zeros(P1,1);               % delta adjustment term for output to hidden weight
                                      % modifications
OldDeltaOut = zeros(P1,1);            % Contains the old delta values to use with the output
                                      % bias units and the momentum term.
NetOut = zeros(P1);                   % Matrix to present summation results to the output
                                      % layer. Only one unit at present.
Oo = zeros(P1);                       % Same dimension as NetOut. Used to store activated
                                      % values of NetOut. Only one unit at present.
Result1 = zeros(depthsize(1,1),1);    % Stores final values at the output layer for each
                                      % pattern. Only one unit at present.
OuterCounter = zeros(epochs,1);
bias = 1;
```

```
iterationcounter = 0;
meanerror = 0;
errorsd = 0;
rms = 0;
relerror = 0;
% SET THE VALUES OF WEIGHTS USING THE RANDOM NO. GENERATOR, EARLY
% STOPPING WEIGHTS OR ENTER THE WEIGHTS MANUALLY
% ----------
prompt1 = '1 - Enter the weights manually ?';
prompt2 = '2 - Let the system automatically apply randomized ones ?';
prompt3 = '3 - Load saved weights for early stopping evaluation on a previously trained
system ?';
disp(prompt1), disp(prompt2), disp(prompt3)
disp(' ')
weightchoice = input('Your choice for the weights i.e. 1, 2 or 3 from above ? ');
if weightchoice == 1
        disp(' ')
        prompt = 'Now enter the weights for the INPUT neurons to the HIDDEN layer ONE
                AT A TIME:';
        disp(prompt)
        disp(' ')
        for j = 1:N1
                prompt1 = 'neuron No.';
                prompt2 = int2str(j);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                for k =1:L1
                        prompt1 = 'weight No.';
                        prompt2 = int2str(k);
                        prompt3 = strcat(prompt1,prompt2);
                        disp(prompt3)
                        Win(j,k) = input('Weight? ')
                end
        end
        disp(' ')
        prompt = 'Now enter the weights for the HIDDEN neurons to the OUTPUT layer
                ONE AT A TIME:';
        disp(prompt)
        disp(' ')
        for j = 1:L1
                prompt1 = 'neuron No.';
                prompt2 = int2str(j);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                for k =1:P1
                        prompt1 = 'weight No.';
                        prompt2 = int2str(k);
                        prompt3 = strcat(prompt1,prompt2);
                        disp(prompt3)
```

```
                Wout(j,k) = input('Weight? ')
            end
        end
        disp(' ')
        prompt = 'Now enter the weights for the BIAS neurons to the HIDDEN layer ONE
                AT A TIME:';
        disp(prompt)
        prompt1 = 'Enter zero if it does not exist';
        disp(prompt1)
        disp(' ')
        for j = 1:L1
            prompt1 = 'bias to hidden neuron No.';
            prompt2 = int2str(j);
            prompt3 = strcat(prompt1,prompt2);
            disp(prompt3)
            BiasHidden(j) = input('Weight? ')
        end
        disp(' ')
        prompt = 'Now enter the weights for the BIAS neurons to the OUTPUT layer ONE
                AT A TIME:';
        disp(prompt)
        prompt1 = 'Enter zero if it does not exist';
        disp(prompt1)
        disp(' ')
        for j = 1:P1
            prompt1 = 'bias to output neuron No.';
            prompt2 = int2str(j);
            prompt3 = strcat(prompt1,prompt2);
            disp(prompt3)
            BiasOut(j) = input('Weight? ')
        end
elseif weightchoice == 2
        Win = randn(N1,L1)/10          % division by 10 needed to reduce instability;
        Wout = randn(L1,P1)/10;
        BiasHidden = randn(L1,1)/10;
        BiasOut = randn(P1,1)/10;
else
        load DepthEarlyStopping OriginalWin OriginalWout OriginalBiasHidden
            OriginalBiasOut;
        Win = OriginalWin;
        Wout = OriginalWout;
        BiasHidden = OriginalBiasHidden;
        BiasOut = OriginalBiasOut;
end
% SAVE ORIGINAL WEIGHTS FOR LATER COMPARISON
% ------------------------------------------------------------------
if weightchoice ~= 3
        OriginalWin = Win;
        OriginalWout = Wout;
```

```
        OriginalBiasHidden = BiasHidden;
        OriginalBiasOut = BiasOut;
end

% PREPROCESSING OF THE DATA
% ---------------------------------------
%  for future development
% SETUP PATTERN AND TARGET MATRICES
% -----------------------------------------------------
Result1 = LatestDepth;
Tg = Result1;
for j = 2:depthsize - 1
        Patterns(j,1)=PreviousDepth(j,1);     % pattern matrix -1st. neuron in input layer
        Patterns(j,2)=PreviousSpeed(j-1,1);   % pattern matrix – 2nd. neuron in input layer
        Patterns(j,3)=PreviousSpeed(j,1);     % pattern matrix -3rd. neuron in input layer
end
% MAIN BODY - START OF LOOP
% ---------------------------------------
for outercounter = 1:epochs
        difference = 0;
        difference1 = 0;
        for innercounter  = 2:depthsize-1              % No. of patterns in the input space
                X = Patterns(innercounter:innercounter,:);
                        % Select a particular pattern for presentation from the input data
                X = X';
                Z = Tg(innercounter:innercounter,:);
                                % Select a particular pattern for testing from the target data
                Z = Z';
                % START OF FORWARD PASS
                % ---------------------------------
                NetIn = Win'*X + BiasHidden;          % net data to the hidden layer
                Oh = NetIn;
                if activation ==1       % Apply activation function to the hidden layer
                        for k = 1:L1
                                Oh(k,1) = 1/(1+exp(-Oh(k,1)));
                        end
                elseif activation ==2
                        for k = 1:L1
                                Oh(k,1) = 2/(1+exp(-2*Oh(k,1)))-1;
                        end
                else
                        for k = 1:L1
                                Oh(k,1) = 1.7159*(2/(1+exp(-4/3*Oh(k,1))))-1);
                        end
                end
                NetOut = Wout'*Oh + BiasOut;
                                % activated data from hidden layer to the output layer
                Oo = NetOut;            % Note: currently only designed for one output unit
```

```
if activation ==1                % Apply activation function to the output layer
        for k = 1:P1
                Oo(k,1) = 1/(1+exp(-Oo(k,1)));
        end
elseif activation ==2
        for k = 1:P1
                Oo(k,1) = 2/(1+exp(-2*Oo(k,1)))-1;
        end
else
        for k = 1:P1
                Oo(k,1) = 1.7159*(2/(1+exp(-4/3*Oo(k,1))))-1);
        end
end
% START OF BACKWARD PASS
% ----------------------------------
ErrorOut = Z - Oo;               % Calculate error signal and Delta
                                 % term for output layer
for j = 1:P1
        if activation ==1
                DeltaOut(j) = ErrorOut(j) * Oo(j)*(1-Oo(j));
                                 % Using sigmoid transfer function
        else
                DeltaOut(j) = ErrorOut(j) * (1-(Oo(j))^2);
                                 % Using tanh transfer function
        end
end
ErrorHidden = Wout*DeltaOut;     % Calculate error signal and Delta
                                 % term for hidden layer
for j = 1:L1
        if activation ==1
                DeltaHidden(j) = Oh(j,1)*(1-Oh(j,1))*ErrorHidden(j);
                                 % Using sigmoid transfer function
        else
                DeltaHidden(j) = (1-(Oh(j,1))^2)*ErrorHidden(j);
                                 % Using tanh transfer function
        end
end
Temp = DeltaOut*Oh';             % Compute Wed and adjust weights
                                 % between hidden and output layers
WedOut = Temp';
Wout = Wout + eta*WedOut +alpha*OldWedOut - decay3*WedOut;
OldWedOut = WedOut;
BiasOut = BiasOut + eta*DeltaOut + alpha*OldDeltaOut - decay4*DeltaOut;
Temp = DeltaHidden*X';           % Compute Wed and adjust weights
                                 % between input and hidden layers
WedIn = Temp';
Win = Win + eta*WedIn +alpha*OldWedIn - decay1*WedIn;
OldWedIn = WedIn;
```

```
            BiasHidden = BiasHidden + eta*DeltaHidden + alpha*OldDeltaHidden –
                decay2*DeltaHidden;
            difference = difference + (Tg(innercounter) - Oo(1))^2;
                                                    % used to calculate the RMS
            difference1 = difference1 + abs(Tg(innercounter) - Oo(1));
                                                % used to calculate the mean error
            Result1(innercounter) = Oo(1);                  % used for plotting
        end
        rms = sqrt(difference/(innercounter-1));
        meanerror = difference1/(innercounter-1);
        subplot(2,1,1),plot (outercounter,factor*rms);      % messy location but needed at
                                            % this point rather than later to save on memory
        OuterCounter(outercounter) = rms;
        hold on;
        if rms < requiredrms
            break
        end
end
% POSTPROCESSING OF THE DATA
% ----------------------------------------
%  for future development!!!
% PLOT THE RMS GRAPH
% ----------------------------
axis ([-inf, inf, 0, 1])
xlabel ('No. of epochs')
ylabel (['RMS x',int2str(factor),' '])
title (['Plot of the Root Mean Square Error v No. of Iterations (Epochs)'])
% PLOT THE DEPTH GRAPH
% --------------------------------
j = [2:depthsize]
        subplot(2,1,2),plot (LatestDepth(j)+ho);    % plot of original water depth along
                                                    % centre of river at different distances
        hold on;
        subplot(2,1,2),plot(Result1(j)+ho,'k');         % plot of the ANN water depth
                                                    % throughout the length of the river
        hold on;
axis ([-inf, inf, -inf, inf])
xlabel ('Distance along centre of river in Km')
ylabel ('Depth in metres')
title (['Water depth above/below still water level'])
% CALCULATION OF MORE STATISTICS
% ------------------------------------------------
for j = 2:depthsize-1
        errorsd = errorsd + ((Result1(j,1)-Tg(j,1)) - meanerror)^2;
        relerror = relerror + abs(Result1(j,1)-Tg(j,1))/abs(Result1(j,1)+ho);
end
errorsd = (errorsd^.5)/(depthsize(1,1)-3);
relerror = relerror*100/(depthsize(1,1)-2);
% SUMMARY OF RESULTS
```

```
% ----------------------------
disp('Summary of the results')
disp('-------------------------------------------------------')
disp('Calculated value(s) of the output layer'), disp(Result1), disp('Target value(s) of the
output layer'),disp(Tg)
disp('rms'),disp(rms),disp('mean error'),disp(meanerror)
disp('error standard deviation '),disp(errorsd), disp('relative percentage error '),disp(relerror)
disp('No. of iterations to satisfy the rms requirement'), disp(outercounter)
disp('Final weight matrix from input layer to hidden layer:'),disp(Win)
disp('Original weight matrix from input layer to hidden layer:'),disp(OriginalWin)
disp('Change from original is:'),disp(Win - OriginalWin)
disp('Final weight matrix from Bias unit(s) to hidden layer:'),disp(BiasHidden)
disp('Original weight matrix from Bias unit(s) to hidden layer:'),disp(OriginalBiasHidden)
disp('Change from original is:'),disp(BiasHidden - OriginalBiasHidden)
disp('Final weight matrix from hidden layer to output layer:'),disp(Wout)
disp('Origiinal weight matrix from hidden layer to output layer:'),disp(OriginalWout)
disp('Change from original is:'),disp(Wout - OriginalWout)
disp('Final weight matrix from Bias unit(s) to output layer:'),disp(BiasOut)
disp('Original weight matrix from Bias unit(s) to output layer:'),disp(OriginalBiasOut)
disp('Change from original is:'),disp(BiasOut - OriginalBiasOut)
disp('-------------------------------------------------------')
% SAVE DATA FOR USE WITH THE VELOCITY ANN PROGRAM AND DEPTH
% VALIDATION PROGRAM
% ------------------------------------------------------------------------------------------
save DepthValidation activation L1 Win Wout BiasHidden BiasOut;
                                        % save for use with the depth validation program
save DepthEarlyStopping OriginalWin OriginalWout OriginalBiasHidden OriginalBiasOut;
                                        % save for use with the early stopping option
save NewDepth Result1;
disp(' ')
s = ' ';                                % must be a single character to hold the result!!!
question = ' ';
question = input('Do you wish to save the rms results to a training file Y/N ? ','s');
switch question
        case 'Y'
                save TrainingRMS OuterCounter;
        case 'y'
                save TrainingRMS OuterCounter;
end
disp(' ')
s = ' ';                                % must be a single character to hold the result!!!
question = ' ';
question = input('Do you wish to save the rms results to a test file Y/N ? ','s');
switch question
        case 'Y'
                save TestRMS OuterCounter;
        case 'y'
                save TestRMS OuterCounter;
end
```

```
% Filename VelocitySolution.m
%       L.H.Rees March 2007
%       USE THIS PROGRAM ALSO FOR VALIDATING DIFFERENT NEURAL
%       NETWORK ARCHITECTURES (DIFFERENT NOS. OF HIDDEN NEURONS).
%       This is a single hidden layer MLP using the sigmoid or tanh activation functions
%       using the backpropogation algorithm, learning rate and momentum terms.
%       Designed for file input of data and incremental updating (not epoch i.e. batch
%       learning). THIS PROGRAM SOLVES FOR THE VELOCITIES. SEE
%       DepthSolution.m FOR THE VERSION THAT SOLVES THE HEIGHTS ABOVE
%       THE STEADY STATE LEVEL. THIS PROGRAM USES THE DATA FILE
%       VenantOriginal to solve for the velocities.
% GET INITIAL DATA
% ----------------------
prompt1 = 'This NN is designed to use three possible types of activation functions';
prompt2 = '1 - The sigmoid (logistic) function';
prompt3 = '2 - The tanh function';
prompt4 = '3 - The LeCun tanh function';
disp(prompt1), disp(prompt2), disp(prompt3), disp(prompt4)
disp(' ')
activation = input('Your choice for the activation function i.e. 1,2 or 3 from above ? ');
disp(' ')
eta = input('Choice for learning rate e.g. usually 0.1 to 1 ? ');
disp(' ')
alpha = input('Value for the momentum term e.g. usually 0.1 to 0.8 ? ');
disp(' ')
decay1 = input('Value for weight decay rate from the input to hidden layer e.g. 0 to 0.005 ?
');
disp(' ')
decay2 = input('Value for weight decay rate from the bias to hidden layer e.g. 0 to 0.005 ? ');
disp(' ')
decay3 = input('Value for weight decay rate from the hidden to output layer e.g. 0 to 0.005 ?
');
disp(' ')
decay4 = input('Value for weight decay rate from the bias to output layer e.g. 0 to 0.005 ? ');
disp(' ')
requiredrms = input('Required value for the root mean square error e.g. 0.01 ? ');
disp(' ')
L1 = input('No. of neurons in the hidden layer e.g. 3 ? ');
disp(' ')
epochs = input('How many epochs i.e. iterations, do you require ? ');
disp(' ')
load VenantOriginal ho speedsize depthsize LatestSpeed PreviousSpeed LatestDepth
PreviousDepth;
load NewDepth Result1;
disp('The pattern of final and penultimate veloocities and depths and also ANN estimates of
the depths from the sister program has now been loaded')
disp(' ')
N1 = 3;                                   % No. of neurons in the input layer
```

```
P1 = 1;                                   % No. of neurons in the output layer
factor = input('Enlargement factor for rms plot e.g. 10 to 100 ? ');
                                          % multiplier to enlarge rms plots since some might not show
% SETUP AND DIMENSIONALIZE ARRAYS
% ------------------------------------------------
BiasHidden = zeros(L1,1);                 % Contains the weights of the bias units for the
                                          % hidden layer. Value of bias is +1
BiasOut = zeros(P1,1);                    % Contains the weights of the bias units for the
                                          % output layer. Value of bias is +1
Patterns = zeros(speedsize(1,1),N1);      % Input data matrix holding all the patterns
Win = zeros(N1,L1);                       % Weights for the input to hidden layers matrix
                                          % Each row separately, must represent all of
                                          % the weights for a given input unit
Wout = zeros(L1,P1);                      % Weights for the hidden to output layers
                                          % Each row separately, must represent all of
                                          % the weights for a given hidden unit
Tg = zeros(speedsize(1,1),P1);            % Target data matrix.
NetIn = zeros(L1,speedsize(1,1));         % Matrix to present summation results to the
                                          % hidden layer
Oh = zeros(N1,L1);                        % Same dimensions as NetIn1. Used to store
                                          % activated values of NetIn1
WedOut = zeros(L1,P1);                    % Weight error derivatives for output to hidden layers.
WedIn = zeros(N1,L1);                     % Weight error derivatives for hidden to input layers.
OldWedOut = zeros(L1,P1);                 % Weight error derivatives for output to hidden layers
                                          % from previous calculation for use with momentum terms.
OldWedIn = zeros(N1,L1);                  % Weight error derivatives for hidden to input layers from
                                          % previous calculation for use with momentum terms
ErrorHidden = zeros(L1,1);                % error signals for hidden to input weight adjustments
DeltaHidden = zeros(L1,1);                % delta adjustment term for hidden to input weight
                                          % modifications
OldDeltaHidden = zeros(L1,1);             % Contains the old delta values to use with the hidden
                                          % bias units and the momentum term.
ErrorOut = zeros(P1,1);                   % error signals for output to hidden weight
                                          % adjustments
DeltaOut = zeros(P1,1);                   % delta adjustment term for output to hidden weight
                                          % modifications
OldDeltaOut = zeros(P1,1);                % Contains the old delta values to use with the output
                                          % bias units and the momentum term.
NetOut = zeros(P1);                       % Matrix to present summation results to the output
                                          % layer. Only one unit at present.
Oo = zeros(P1);                           % Same dimension as NetOut. Used to store activated
                                          % values of NetOut. Only one unit at present.
Result2 = zeros(speedsize(1,1),1);        % Stores final values at the output layer for each
                                          % pattern. Only one unit at present.
OuterCounter = zeros(epochs,1);
bias = 1;
iterationcounter = 0;
meanerror = 0;
errorsd = 0;
```

```
rms = 0;
relerror = 0;
% SET THE VALUES OF WEIGHTS USING THE RANDOM NO. GENERATOR, EARLY
 % STOPPING WEIGHTS OR ENTER THE WEIGHTS MANUALLY
% ----------------------------------------------------------------------------------------------------
prompt1 = '1 - Enter the weights manually ?';
prompt2 = '2 - Let the system automatically apply randomized ones ?';
prompt3 = '3 - Load saved weights for early stopping evaluation on a previously trained
system ?';
disp(prompt1), disp(prompt2), disp(prompt3)
disp(' ')
weightchoice = input('Your choice for the weights i.e. 1, 2 or 3 from above ? ');
if weightchoice == 1
        disp(' ')
        prompt = 'Now enter the weights for the INPUT neurons to the HIDDEN layer ONE
            AT A TIME:';
        disp(prompt)
        disp(' ')
        for j = 1:N1
                prompt1 = 'neuron No.';
                prompt2 = int2str(j);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                for k =1:L1
                        prompt1 = 'weight No.';
                        prompt2 = int2str(k);
                        prompt3 = strcat(prompt1,prompt2);
                        disp(prompt3)
                        Win(j,k) = input('Weight? ')
                end
        end
        disp(' ')
        prompt = 'Now enter the weights for the HIDDEN neurons to the OUTPUT layer
                    ONE AT A TIME:';
        disp(prompt)
        disp(' ')
        for j = 1:L1
                prompt1 = 'neuron No.';
                prompt2 = int2str(j);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                for k =1:P1
                        prompt1 = 'weight No.';
                        prompt2 = int2str(k);
                        prompt3 = strcat(prompt1,prompt2);
                        disp(prompt3)
                        Wout(j,k) = input('Weight? ')
                end
        end
```

```
        disp(' ')
        prompt = 'Now enter the weights for the BIAS neurons to the HIDDEN layer ONE
                  AT A TIME:';
        disp(prompt)
        prompt1 = 'Enter zero if it does not exist';
        disp(prompt1)
        disp(' ')
        for j = 1:L1
                prompt1 = 'bias to hidden neuron No.';
                prompt2 = int2str(j);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                BiasHidden(j) = input('Weight? ')
        end
        disp(' ')
        prompt = 'Now enter the weights for the BIAS neurons to the OUTPUT layer ONE
                  AT A TIME:';
        disp(prompt)
        prompt1 = 'Enter zero if it does not exist';
        disp(prompt1)
        disp(' ')
        for j = 1:P1
                prompt1 = 'bias to output neuron No.';
                prompt2 = int2str(j);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                BiasOut(j) = input('Weight? ')
        end
elseif weightchoice == 2
        Win = randn(N1,L1)/10;          % divison by 10 needed to reduce instability
        Wout = randn(L1,P1)/10;
        BiasHidden = randn(L1,1)/10;
        BiasOut = randn(P1,1)/10;
else
        load VelocityEarlyStopping OriginalWin OriginalWout OriginalBiasHidden
            OriginalBiasOut;
        Win = OriginalWin;
        Wout = OriginalWout;
        BiasHidden = OriginalBiasHidden;
        BiasOut = OriginalBiasOut;
end
% SAVE ORIGINAL WEIGHTS FOR LATER COMPARISON
% ------------------------------------------------------------------
if weightchoice ~= 3
        OriginalWin = Win;
        OriginalWout = Wout;
        OriginalBiasHidden = BiasHidden;
        OriginalBiasOut = BiasOut;
end
```

```
% PREPROCESSING OF THE DATA
% ----------------------------------------
%  for future development
% SETUP PATTERN AND TARGET MATRICES
% ------------------------------------------------------
Tg = LatestSpeed;
for j = 1:depthsize-1
        Patterns(j,1)= PreviousSpeed(j);
        Patterns(j,2)= Result1(j);
        Patterns(j,3) = Result1(j+1);
end
% MAIN BODY - START OF LOOP
% ------------------------------------
for outercounter = 1:epochs
        difference = 0;
        difference1 = 0;
        for innercounter  = 1:speedsize              % No. of patterns in the input space
                X = Patterns(innercounter:innercounter,:);   % Select a particular pattern
                                                     % for presentation from input data
                X = X';
                Z = Tg(innercounter:innercounter,:);         % Select a particular pattern
                                                     % for testing from the target data
                Z = Z';
                % START OF FORWARD PASS
                % ---------------------------------
                NetIn = Win'*X + BiasHidden;         % Present net data to hidden layer
                Oh = NetIn;                  % Apply activation function to hidden layer
                if activation ==1
                        for k = 1:L1
                                Oh(k,1) = 1/(1+exp(-Oh(k,1)));
                        end
                elseif activation ==2
                        for k = 1:L1
                                Oh(k,1) = 2/(1+exp(-2*Oh(k,1)))-1;
                        end
                else
                        for k = 1:L1
                                Oh(k,1) = 1.7159*(2/(1+exp(-4/3*Oh(k,1)))-1);
                        end
                end
                NetOut = Wout'*Oh + BiasOut;         % Present activated data from hidden
                                                     % layer to the output layer
                Oo = NetOut;                 % Note: currently only designed for one
                                             % output unit
                if activation ==1            % Apply activation function to output layer
                        for k = 1:P1
                                Oo(k,1) = 1/(1+exp(-Oo(k,1)));
                        end
```

```
elseif activation ==2
        for k = 1:P1
                Oo(k,1) = 2/(1+exp(-2*Oo(k,1)))-1;
        end
else
        for k = 1:P1
                Oo(k,1) = 1.7159*(2/(1+exp(-4/3*Oo(k,1))))-1);
        end
end
% START OF BACKWARD PASS
% ----------------------------------
ErrorOut = Z - Oo;              % Calculate error signal and Delta term
                                % for  output layer
for j = 1:P1
        if activation ==1
                DeltaOut(j) = ErrorOut(j) * Oo(j)*(1-Oo(j));
                                % Using sigmoid transfer function
        else
                DeltaOut(j) = ErrorOut(j) * (1-(Oo(j))^2);
                                % Using tanh transfer function
        end
end
ErrorHidden = Wout*DeltaOut;    % Calculate error signal and Delta
                                % term for hidden layer
for j = 1:L1
        if activation ==1
                DeltaHidden(j) = Oh(j,1)*(1-Oh(j,1))*ErrorHidden(j);
                                % Using sigmoid transfer function
        else
                DeltaHidden(j) = (1-(Oh(j,1))^2)*ErrorHidden(j);
                                % Using tanh transfer function
        end
end
Temp = DeltaOut*Oh';            % Compute Wed and adjust weights
                                % between hidden and output layers
WedOut = Temp';
Wout = Wout + eta*WedOut +alpha*OldWedOut - decay3*WedOut;
OldWedOut = WedOut;
BiasOut = BiasOut + eta*DeltaOut + alpha*OldDeltaOut - decay4*DeltaOut;
Temp = DeltaHidden*X';          % Compute Wed and adjust weights
                                % between input and hidden layers
WedIn = Temp';
Win = Win + eta*WedIn +alpha*OldWedIn - decay1*WedIn;
OldWedIn = WedIn;
BiasHidden = BiasHidden + eta*DeltaHidden + alpha*OldDeltaHidden –
    decay2*DeltaHidden;
difference = difference + (Tg(innercounter) - Oo(1))^2;
                                % used to calculate the RMS
difference1 = difference1 + abs(Tg(innercounter) - Oo(1));
```

```
                                        % used to calculate the mean error
            Result2(innercounter) = Oo(1);      % used for plotting
        end
        rms = sqrt(difference/innercounter);
        meanerror = difference1/innercounter;
        subplot(2,1,1),plot(outercounter,factor*rms);% messy location but needed a
                                        % this point rather than later to save on memory.
        OuterCounter(outercounter) = rms;
        hold on;
        if rms < requiredrms
            break
        end
end
% POSTPROCESSING OF THE DATA
% ------------------------------------------
% for future development!!!
% PLOT THE RMS GRAPH
% ----------------------------
axis ([-inf, inf, 0, 1])
xlabel ('No. of epochs')
ylabel (['RMS x',int2str(factor),' '])
title (['Plot of the Root Mean Square Error v No. of Iterations (Epochs)'])
% PLOT THE VELOCITY GRAPH
% ----------------------------------
j = [1:speedsize]
        subplot(2,1,2),plot (LatestSpeed(j));       % plot of original water velocities
                                        % along centre of river at different distances
        hold on;
        subplot(2,1,2),plot(Result2(j),'k');        % plot of the ANN water velocities
                                        % throughout the length of the river
        hold on;
axis ([-inf, inf, -inf, inf])
xlabel ('Distance along centre of river in Km')
ylabel ('Velocity in metres per sec.')
title (['Depth averaged velocities'])
% CALCULATION OF MORE STATISTICS
% ------------------------------------------------
for j = 1:speedsize
        errorsd = errorsd + ((Result2(j,1)-Tg(j,1)) - meanerror)^2;
        relerror = relerror + abs(Result2(j,1)-Tg(j,1))/abs(Result2(j,1)+ho);
end
errorsd = (errorsd^.5)/(speedsize(1,1)-1);
relerror = relerror*100/speedsize(1,1);
% SUMMARY OF RESULTS
% ----------------------------
disp('Summary of the results')
disp('-------------------------------------------------------')
disp('Calculated value(s) of the output layer'), disp(Result2), disp('Target value(s) of the
output layer'),disp(Tg)
```

```
disp('rms'),disp(rms),disp('mean error'),disp(meanerror)
disp('error standard deviation '),disp(errorsd), disp('relative percentage error '),disp(relerror)
disp('No. of iterations to satisfy the rms requirement'), disp(outercounter)
disp('Final weight matrix from input layer to hidden layer:'),disp(Win)
disp('Original weight matrix from input layer to hidden layer:'),disp(OriginalWin)
disp('Change from original is:'),disp(Win - OriginalWin)
disp('Final weight matrix from Bias unit(s) to hidden layer:'),disp(BiasHidden)
disp('Original weight matrix from Bias unit(s) to hidden layer:'),disp(OriginalBiasHidden)
disp('Change from original is:'),disp(BiasHidden - OriginalBiasHidden)
disp('Final weight matrix from hidden layer to output layer:'),disp(Wout)
disp('Origiinal weight matrix from hidden layer to output layer:'),disp(OriginalWout)
disp('Change from original is:'),disp(Wout - OriginalWout)
disp('Final weight matrix from Bias unit(s) to output layer:'),disp(BiasOut)
disp('Original weight matrix from Bias unit(s) to output layer:'),disp(OriginalBiasOut)
disp('Change from original is:'),disp(BiasOut - OriginalBiasOut)
disp('------------------------------------------------------------')
% SAVE DATA FOR USE WITH THE VELOCITY VALIDATION ANN
% PROGRAM AND TestRMS
% ------------------------------------------------------------------------
save VelocityValidation activation L1 Win Wout BiasHidden BiasOut;
                                        % save for use with the velocity validation program
save VelocityEarlyStopping OriginalWin OriginalWout OriginalBiasHidden
OriginalBiasOut;
save NewSpeed Result2;
disp(' ')
s = ' ';                                % must be a single character to hold the result!!!
question = ' ';
question = input('Do you wish to save the rms results to a training file Y/N ? ','s');
switch question
        case 'Y'
                save TrainingRMS OuterCounter;
        case 'y'
                save TrainingRMS OuterCounter;
end
disp(' ')
s = ' ';                                % must be a single character to hold the result!!!
question = ' ';
question = input('Do you wish to save the rms results to a test file Y/N ? ','s');
switch question
        case 'Y'
                save TestRMS OuterCounter;
        case 'y'
                save TestRMS OuterCounter;
end
```

```
% Filename CombinedNetwork.m
%       L.H.Rees March 2007
%       This program is used to display the depths and velocities of the combined network.
%        It works independently of the FDS and training programs.
%       All that is needed to start it off is some initial data from the
%        VenantDataGenerator2 program.
% GET INITIAL DATA
% ----------------------
load VenantOriginal ho speedsize depthsize LatestSpeed PreviousSpeed LatestDepth
      PreviousDepth;                          % initial data to start it off
load DepthValidation activation L1 Win Wout BiasHidden BiasOut;
                          % load fixed weights from the depth training phase.
DepthWin = Win;
DepthWout = Wout;
DepthBiasHidden = BiasHidden;
DepthBiasOut = BiasOut;
load VelocityValidation activation L1 Win Wout BiasHidden BiasOut;
                          % load fixed weights from the velocity training program
VelocityWin = Win;
VelocityWout = Wout;
VelocityBiasHidden = BiasHidden;
VelocityBiasOut = BiasOut;
N1 = 3;                                   % No. of neurons in input layer
P1  = 1;                                  % No. of neurons in output layer
Patterns1 = zeros(depthsize-2,N1);        % Input data matrix holding all patterns
                                          % for depth network
Patterns2 = zeros(speedsize,N1);          % Input data matrix holding all patterns
                                          % for velocity network
Tg1 = zeros(depthsize,P1);                % Target data matrix for depth network
Tg2 = zeros(speedsize,P1);                % Target data matrix for velocity network
Result2 = zeros(speedsize,1);             % Stores final values at the output layer
                                          % for each pattern. Only one unit at present.
NetIn1 = zeros(L1,depthsize-2);           % Matrix to present summation results to
                                          % the hidden layer
Oh1 = zeros(N1,L1);                       % Same dimensions as NetIn1. Used to
                                          % store activated values of NetIn1
NetOut1 = zeros(P1);                      % Matrix to present summation results to
                                          % the output layer. Only one unit at present.
Oo1 = zeros(P1);                          % Same dimension as NetOut. Used to
                                          % store activated values of NetOut
                                          % Only one unit at present.
NetIn2 = zeros(L1,speedsize);             % Matrix to present summation results to
                                          % the hidden layer
Oh2 = zeros(N1,L1);                       % Same dimensions as NetIn1. Used to
                                          % store activated values of NetIn1
NetOut2 = zeros(P1);                      % Matrix to present summation results to
                                          % the output layer. Only one unit at present.
Oo2 = zeros(P1);                          % Same dimension as NetOut. Used to
```

```
                                        % store activated values of NetOut.
                                        % Only one unit at present.
depthmeanerror = 0;
depthrms = 0;
deptherrorsd = 0;
depthrelerror = 0;
velocitymeanerror = 0;
velocityrms = 0;
velocityerrorsd = 0;
velocityrelerror = 0;
%START OF CALCULATIONS
%-----------------------------------
% SETUP PATTERN AND TARGET MATRICES FOR DEPTH UPDATING
% ------------------------------------------------------------------------------
Result1 = LatestDepth;
Tg1 = LatestDepth;
Result2 = LatestSpeed;
Tg2 = LatestSpeed;
for j = 2:depthsize-1
        Patterns1(j,1)=PreviousDepth(j,1);      % complete pattern matrix – first
                                                % neuron in input layer
        Patterns1(j,2)=PreviousSpeed(j-1,1);    % complete pattern matrix – second
                                                % neuron in input layer
        Patterns1(j,3)=PreviousSpeed(j,1);      % complete pattern matrix – third
                                                % neuron in input layer
end
% START OF LOOP TO UPDATE DEPTHS
% ----------------------------------------------
depthdifference = 0;
for j = 2:depthsize-1                           % No. of patterns in the input space
        X1 = Patterns1(j:j,:);                  % Select a particular pattern for presentation
                                                % from the input data
        X1 = X1';
        NetIn1 = DepthWin'*X1 + DepthBiasHidden;    % Present the net data to the
                                                    % hidden layer
        Oh1 = NetIn1;                               % Apply activation function to
                                                    % the hidden layer
        if activation ==1
                for k = 1:L1
                        Oh1(k,1) = 1/(1+exp(-Oh1(k,1)));
                end
        elseif activation ==2
                for k = 1:L1
                        Oh1(k,1) = 2/(1+exp(-2*Oh1(k,1)))-1;
                end
        else
                for k = 1:L1
                        Oh1(k,1) = 1.7159*(2/(1+exp(-4/3*Oh1(k,1)))-1);
                end
```

```
        end
        NetOut1 = DepthWout'*Oh1 + DepthBiasOut;        % Present activated data from
                                                        % hidden layer to the output layer
        Oo1 = NetOut1;                                  % currently only designed for one output unit
        if activation ==1                               % Apply activation function to the output layer
                for k = 1:P1
                        Oo1(k,1) = 1/(1+exp(-Oo1(k,1)));
                end
        elseif activation ==2
                for k = 1:P1
                        Oo1(k,1) = 2/(1+exp(-2*Oo1(k,1)))-1;
                end
        else
                for k = 1:P1
                        Oo1(k,1) = 1.7159*(2/(1+exp(-4/3*Oo1(k,1)))-1);
                end
        end
        depthdifference = depthdifference + abs(Tg1(j) - Oo1(1));   % used to calculate
                                                                    % the mean error
        Result1(j) = Oo1(1);
end
% CALCULATION OF DEPTH STATISTICS
% ------------------------------------------------
depthmeanerror = depthdifference/depthsize(1,1);
for j = 1:depthsize
        depthrms = depthrms + (Result1(j) - Tg1(j))^2;          % NOT rms per epoch
        deptherrorsd = deptherrorsd + ((Result1(j)-Tg1(j)) - depthmeanerror)^2;
        depthrelerror = depthrelerror + abs(Result1(j)-Tg1(j))/abs(Result1(j)+ho);
end
deptherrorsd = (deptherrorsd^.5)/(depthsize(1,1)-1);
depthrelerror = depthrelerror*100/depthsize(1,1);
depthrms = (depthrms/depthsize(1,1))^.5;
% SETUP PATTERN AND TARGET MATRICES FOR VELOCITY UPDATING
% ------------------------------------------------------------------------------------
for j = 1:depthsize-1
        Patterns2(j,1)= PreviousSpeed(j);
        Patterns2(j,2)= Result1(j);                             %LatestDepth(j)
        Patterns2(j,3) = Result1(j+1);                          %LatestDepth(j+1)
end
% START OF LOOP TO UPDATE VELOCITIES
% ------------------------------------------------
velocitydifference = 0;
for j = 1:speedsize                             % No. of patterns in the input space
        X2 = Patterns2(j:j,:);                  % Select a particular pattern for
                                                % presentation from the input data
        X2 = X2';
        NetIn2 = VelocityWin'*X2 + VelocityBiasHidden;      % Present the net data
                                                            % to the hidden layer
```

```
        Oh2 = NetIn2;
        if activation ==1                % Apply activation function to the hidden layer
                for k = 1:L1
                        Oh2(k,1) = 1/(1+exp(-Oh2(k,1)));
                end
        elseif activation ==2
                for k = 1:L1
                        Oh2(k,1) = 2/(1+exp(-2*Oh2(k,1)))-1;
                end
        else
                for k = 1:L1
                        Oh2(k,1) = 1.7159*(2/(1+exp(-4/3*Oh2(k,1))))-1);
                end
        end
        NetOut2 = VelocityWout'*Oh2 + VelocityBiasOut;          % Present activated
                                        % data from hidden layer to the output layer
        Oo2 = NetOut2;                  % currently only designed for one output unit
        if activation ==1               % Apply activation function to the output layer
                for k = 1:P1
                        Oo2(k,1) = 1/(1+exp(-Oo2(k,1)));
                end
        elseif activation ==2
                for k = 1:P1
                        Oo2(k,1) = 2/(1+exp(-2*Oo2(k,1)))-1;
                end
        else
                for k = 1:P1
                        Oo2(k,1) = 1.7159*(2/(1+exp(-4/3*Oo2(k,1))))-1);
                end
        end
        velocitydifference = velocitydifference + abs(Tg2(j) - Oo2(1));
                                                % used to calculate the mean error
        Result2(j) = Oo2(1);
end
% CALCULATION OF VELOCITY STATISTICS
% ----------------------------------------------------
velocitymeanerror = velocitydifference/speedsize(1,1);
for j = 1:speedsize
        velocityrms = velocityrms + (Result2(j) - Tg2(j))^2;          % NOT rms per epoch
        velocityerrorsd = velocityerrorsd + ((Result2(j)-Tg2(j)) - velocitymeanerror)^2;
        velocityrelerror = velocityrelerror + abs(Result2(j)-Tg2(j))/abs(Result2(j)+ho);
end
velocityerrorsd = (velocityerrorsd^.5)/(speedsize(1,1)-1);
velocityrelerror = velocityrelerror*100/speedsize(1,1);
velocityrms = (velocityrms/speedsize(1,1))^.5;
% PLOT THE DEPTH GRAPH
% --------------------------------
j = [2:depthsize]
```

```
        subplot(2,1,1), plot (Result1(j)+ho,'k-');      % plot of ANN depth along centre
                                                        %  of river at different distances
        hold on;
        subplot(2,1,1),plot(Tg1(j)+ho);                 % plot of target depth throughout
                                                        % the length of the river
        hold on;
axis ([-inf, inf, -inf, inf])
xlabel ('Distance along centre of river in Km')
ylabel ('Water depth in metres')
title (['Water depth'])
% PLOT THE VELOCITY GRAPH
% ----------------------------------
j = [1:speedsize]
        subplot(2,1,2), plot (Result2(j),'k-');         % plot of ANN velocity along centre
                                                        % of river at different distances
        hold on;
        subplot(2,1,2),plot(Tg2(j));                    % plot of the target velocity throughout
                                                        % the length of the river
        hold on;
axis([-inf inf -inf inf])
xlabel ('Distance along centre of river in Km')
ylabel ('Velocity in metres per sec')
title (['Velocity'])
% SUMMARY OF RESULTS
% ------------------------------
disp('Summary of the depth results')
disp('-------------------------------------------------------------------------------------------')
disp('mean error of the depth calculations '),disp(depthmeanerror),disp('rms of the depth
calculations '),disp(depthrms)
disp('error standard deviation of the depth calculations '),disp(deptherrorsd),disp('relative
percentage error of the depth calculations '),disp(depthrelerror)
disp(' ')
disp('Summary of the velocity results')
disp('-------------------------------------------------------------------------------------------')
disp('mean error of the velocity calculations '),disp(velocitymeanerror),disp('rms of the
velocity calculations '),disp(velocityrms)
disp('error standard deviation of the velocity calculations
'),disp(velocityerrorsd),disp('relative percentage error of the velocity calculations
'),disp(velocityrelerror)
disp(' ')
disp('-------------------------------------------------------------------------------------------')
```

```
% Filename DepthWeightFileCreation.m
% L.H.Rees March 2007
% This program is designed for entering weights etc for the depth program
% for validation.
% ------------------------------------------------------------------------------------------
prompt1 = 'This NN is designed to use three possible types of activation functions';
prompt2 = '1 - The sigmoid (logistic) function';
prompt3 = '2 - The tanh function';
prompt4 = '3 - The LeCun tanh function';
disp(prompt1), disp(prompt2), disp(prompt3), disp(prompt4)
disp(' ')
activation = input('Your choice for the activation function i.e. 1, 2 or 3 from above ? ');
disp(' ')
L1 = input('No. of neurons in the hidden layer e.g. 3 ? ');
disp(' ')
load VenantOriginal ho speedsize depthsize LatestSpeed PreviousSpeed LatestDepth
PreviousDepth;
disp('The pattern of final and penultimate veloocities and depths has now been loaded')
disp(' ')
M1 = depthsize(1,1)-2;          % No. of patterns. Two less because cannot include
                                % velocities outside of the two boundaries
N1 = 3;                         % No. of neurons in the input layer
P1 = 1;                         % No. of neurons in the output layer
% SETUP AND DIMENSIONALIZE ARRAYS
% ------------------------------------------------
BiasHidden = zeros(L1,1);       % Contains the weights of the bias units for the
                                % hidden layer. Value of bias is +1
BiasOut = zeros(P1,1);          % Contains the weights of the bias units for the
                                % output layer. Value of bias is +1
Win = zeros(N1,L1);             % Weights for the input to hidden layers matrix
                                % Each row separately, must represent all of the
                                % weights for a given input unit
Wout = zeros(L1,P1);            % Weights for the hidden to output layers matrix
                                % Each row separately, must represent all of the
                                % weights for a given hidden unit
% SET THE VALUES OF WEIGHTS USING THE RANDOM NO. GENERATOR, EARLY
% STOPPING WEIGHTS OR ENTER THE WEIGHTS MANUALLY
%-------------------------------------------------------------------------------------------
disp(' ')
prompt = 'Now enter the weights for the INPUT neurons to the HIDDEN layer ONE AT A
TIME:';
disp(prompt)
disp(' ')
for j = 1:N1
        prompt1 = 'neuron No.';
        prompt2 = int2str(j);
        prompt3 = strcat(prompt1,prompt2);
        disp(prompt3)
```

```
        for k =1:L1
                prompt1 = 'weight No.';
                prompt2 = int2str(k);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                Win(j,k) = input('Weight? ')
        end
end
disp(' ')
prompt = 'Now enter the weights for the HIDDEN neurons to the OUTPUT layer ONE AT
        A TIME:';
disp(prompt)
disp(' ')
for j = 1:L1
        prompt1 = 'neuron No.';
        prompt2 = int2str(j);
        prompt3 = strcat(prompt1,prompt2);
        disp(prompt3)
        for k =1:P1
                prompt1 = 'weight No.';
                prompt2 = int2str(k);
                prompt3 = strcat(prompt1,prompt2);
                disp(prompt3)
                Wout(j,k) = input('Weight? ')
        end
end
disp(' ')
prompt = 'Now enter the weights for the BIAS neurons to the HIDDEN layer ONE AT A
        TIME:';
disp(prompt)
prompt1 = 'Enter zero if it does not exist';
disp(prompt1)
disp(' ')
for j = 1:L1
        prompt1 = 'bias to hidden neuron No.';
        prompt2 = int2str(j);
        prompt3 = strcat(prompt1,prompt2);
        disp(prompt3)
        BiasHidden(j) = input('Weight? ')
end
disp(' ')
prompt = 'Now enter the weights for the BIAS neurons to the OUTPUT layer ONE AT A
        TIME:';
disp(prompt)
prompt1 = 'Enter zero if it does not exist';
disp(prompt1)
disp(' ')
for j = 1:P1
        prompt1 = 'bias to output neuron No.';
```

```
        prompt2 = int2str(j);
        prompt3 = strcat(prompt1,prompt2);
        disp(prompt3)
        BiasOut(j) = input('Weight? ')
end
% SAVE DATA FOR USE WITH THE VELOCITY ANN PROGRAM AND DEPTH
% VALIDATION PROGRAM
% ------------------------------------------------------------------------
save DepthValidation activation L1 Win Wout BiasHidden BiasOut;
```

Note: a companion program to this one allows creation of weights for the velocity validation program. It is identical in all respects to the depth one save for the last line of code above which should read:

'save VelocityValidation activation L1 Win Wout BiasHidden BiasOut;'

```
% Filename RMSTest.m
%       L.H.Rees April 2007
%       This program compares the RMS from the training data against the RMS STORED
%       IN THE FILE TestRMS.
% ------------------------------------------------------------------------------------------
factor = input('Vertical scale factor e.g. 100? ');
load TrainingRMS OuterCounter;
for j = 1:size(OuterCounter)
        rms = OuterCounter(j);
        plot (j,rms*factor);
        hold on;
end
disp('Training RMS = ');
disp(rms);
load TestRMS OuterCounter;
for j = 1:size(OuterCounter)
        rms = OuterCounter(j);
        plot (j,rms*factor);
        hold on;
end
disp('Test RMS = ');
disp(rms);
axis ([-inf, inf, 0, 1])
xlabel ('No. of epochs')
ylabel (['RMS x ',int2str(factor),' '])
title (['Plot of the Root Mean Square Error v No. of Iterations (Epochs)'])
```

Appendix D
Support Vector Machines – an alternative paradigm

D.1 Introduction

Mathematically, support vector machines (SVMs) are a range of classification and regression algorithms formulated from the principles of Statistical Learning Theory developed by Vapnik [105]. They have been applied successfully to classification tasks such as pattern recognition, OCR and more recently, to regression and time series. In recent years, a number of non-linear classification and regression variants of SVMs have been developed and these have been benchmarked against artificial neural networks (ANNs). It has been found that the empirical performance of SVMs is generally as good as the best ANN solutions. It has been hypothesised, that this is because there are fewer model parameters to optimise in the SVM approach so that the latter is not so prone to 'over fitting', a situation that arises when there is insufficient data for training. However, Han et. al. [40] (cf. Chapter 3), found that SVMs did suffer from overfitting and underfitting as much as conventional neural network models.

In comparison to physically based hydrodynamic models, SVMs are data parsimonious, a property that Asefa et. al. [6], noted. Support vector machines are state-of-the art machine learning methodology with some very important features. Using a hypothesis space of linear functions in a kernel induced higher dimensional feature space, the SVMs are trained using a learning algorithm from optimization theory. The principle of SVMs is to minimize the generalized model error (risk) rather than aiming to minimize the mean square error over a training set. Further, because of Mercer's condition [78] on the kernels, the corresponding optimization problems are convex so that local minima are avoided. However, SVMs do have their drawbacks. The selection of suitable kernels as well as hyper-parameters is currently still rather heuristic, and as a consequence therefore problematic, a difficulty referred to by Han et. al. [40]. As for artificial neural networks, support vector machines can be represented as a one hidden layer network where the weights from the input to hidden layer are non-linear and those from the hidden to output layer are linear.

However, a major distinction between the two techniques is with regard to the choice of parameters in the training algorithm. Using gradient or clustering-based approaches, neural networks generally adapt all of the parameters whilst in contrast, support vector machines choose the parameters for the first layer to be the training input vectors thus minimising the VC-dimension, Vapnik and Chervenenkis [104]. SVMs contain a large class of certain neural networks, RBF networks and polynomial classifiers as special cases. The discussion in this appendix will therefore concentrate in order on: support vector learning, hyperplane classifiers; features spaces and kernels; support vector machines; implementation and historical development.

D.2 Support Vector Learning

The support vector algorithm can be analysed mathematically because it has a correspondence with a linear method in a high dimensional feature space that itself is non-linearly related to the input space. In practice, the computations do not take place in the feature space but instead, using *kernels*, all the necessary computations are performed directly in the input space.

The objective here is to estimate/find a function $f : \mathbb{R}^n \to \{\pm 1\}$ using input-output training data $(\mathbf{x}_1, y_1)...(\mathbf{x}_n, y_n) \in \mathbb{R}^n \times \{\pm 1\}$ such that new (unseen) data sets will be correctly classified by the function f. That is, $f(\mathbf{x}_i) = y_i, \forall_{new\ (x_i, y_i)}$. It is assumed here that the new data belongs to the same population (that is generated from the same underlying probability distribution $P(\mathbf{x}, y)$) as the training data.

Noting that if the class of functions, from which the estimated required f, is not restricted, then even if this estimated function f performs well on the training data, it may not necessarily generalise well to unseen (test) examples. In other words, since only the training data is available, it is not possible to determine which one of a set of functions is the best choice. Thus minimisation of the training error (sometimes called *empirical risk*) alone using

$$R_{emp}|f| = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}|f(\mathbf{x}_i) - y_i| \qquad \textbf{(D1)}$$

will not necessarily imply a small expected test error (called risk) where the latter is defined by the indefinite integral

$$R|f| = \int \frac{1}{2}|f(\mathbf{x}) - y| dP(\mathbf{x}, y) \qquad \textbf{(D2)}$$

In fact, *statistical learning theory,* Vapnik [105], dictates that it is vital to restrict the class of functions to one with a *capacity* that is suitable for the amount of available training data. In fact, VC theory provides bounds on the test error, the minimization of which [bounds] depends on both the empirical risk and the capacity of the function class.

D.3 Hyperplane classifiers

It appears then that to design suitable learning algorithms, it is necessary to find a class of functions for which the capacity can be computed. Some researchers (including Vapnik) therefore considered the class of hyperplanes $(\mathbf{w}^T \bullet \mathbf{x}) + b = 0,\ \mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R}$ that corresponds to decision functions of form

$$f(\mathbf{x}) = \text{sgn}((\mathbf{w}^T \bullet \mathbf{x}) + b) \qquad \textbf{(D3)}$$

for which it would be possible to construct the function f for a given separable problem, that is, one where it is possible to distinguish between two classes of data. It can be seen that amongst all the possible hyperplanes separating two classes, the one leading to a maximum margin of separation between the two classes is unique and is hence referred to as the *optimal hyperplane:*

$$\max_{\mathbf{w},b} \left\{ \min \left\{ \|\mathbf{x} - \mathbf{x}_i\| : (\mathbf{w}^T \bullet \mathbf{x}) + b = 0, \quad i \in \mathbb{N}^+, \mathbf{x}, \mathbf{w} \in \mathbb{R}^n, b \in \mathbb{R} \right\} \right\} \qquad \textbf{(D4)}$$

where, the larger the margin size, the smaller the required capacity. Continuing,

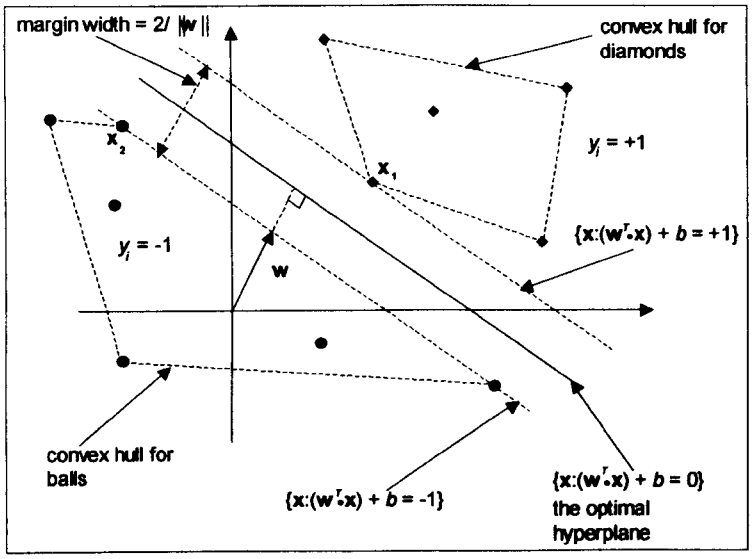

Figure D1: The optimal hyperplane

using the arguments of Burges [16] and Scholkopf et. al. [94], consider a simple separable (binary) classification problem on separation of toy balls from toy diamonds as depicted in fig. D1. It can be seen that the optimal hyperplane is orthogonal to the shortest line connecting the convex hulls of the two classes. Further, it intersects this line at the mid-point. There is a weight vector **w** and a threshold value b such that $y_i((\mathbf{w}^T \bullet \mathbf{x}_i) + b) > 0$. If now **w** and b are rescaled, so that point(s) closest to the hyperplane, in the fig. D1 $\mathbf{x}_1$ and $\mathbf{x}_2$ for instance, satisfy $\left|(\mathbf{w}^T \bullet \mathbf{x}_i) + b\right| = 1$, then a closed (canonical) form (**w**, b) for the hyperplane is obtained of form

$$y_i((\mathbf{w}^T \bullet \mathbf{x}_i) + b) \geq 1 \qquad \textbf{(D5)}$$

Since

$$\left(\mathbf{w}^T \bullet \mathbf{x}_1\right) + b = +1 \quad \text{and} \quad \left(\mathbf{w}^T \bullet \mathbf{x}_2\right) + b = -1$$

$$\text{this} \Rightarrow \left(\mathbf{w}^T \bullet (\mathbf{x}_1 - \mathbf{x}_2)\right) = 2 \qquad \textbf{(D6)}$$

$$\text{which } \therefore \Rightarrow \left(\left(\frac{\mathbf{w}}{\|\mathbf{w}\|}\right)^T \bullet (\mathbf{x}_1 - \mathbf{x}_2)\right) = \frac{2}{\|\mathbf{w}\|}$$

the latter result describing the width of the margin as measured perpendicularly to the hyperplane. Hence, to maximize this margin, it is necessary to minimize $\|\mathbf{w}\|$ subject to equation **(D5)** which is in effect, a *constrained quadratic optimisation problem.*

Following the method of Scholkopf et. al. [94], introduce the Lagrange multipliers $\{\alpha_i : \alpha_i \geq 0\}$ which gives rise to the Lagrangian operator of form:

$$L(\mathbf{w},b,\boldsymbol{\alpha}) = \frac{\|\mathbf{w}\|^2}{2} - \sum_{i=1}^{n} \alpha_i (y_i((\mathbf{w}^T \bullet \mathbf{x}_i) + b) - 1) \qquad \textbf{(D7)}$$

where L has to be minimized with respect to the primal variables **w** and b and maximized with respect to the dual variables α_i (the Lagrange multipliers). This effectively means looking for a saddle point at which $\frac{\partial L}{\partial b}(\mathbf{w},b,\boldsymbol{\alpha}) = \frac{\partial L}{\partial \mathbf{w}}(\mathbf{w},b,\boldsymbol{\alpha}) = 0$.

Since $\|\mathbf{w}\|^2 \equiv \mathbf{w} \bullet \mathbf{w}$, differentiating equation **(D7)** with respect to these conditions gives

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \text{ and } \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \,. \qquad \textbf{(D8)}$$

From equations **(D8)**, it can be seen that the solution vector **w** to the optimal hyperplane is therefore comprised of a subset of the training patterns, that is, those patterns for which $\alpha_i \neq 0$. This subset of patterns $\{\mathbf{x}_i\}$ that lie on the margin, such as $\mathbf{x}_1$ and $\mathbf{x}_2$ in fig. D1 are the so-called **support vectors** and they contain all the relevant information about the classification problem. The hyperplane is determined completely by the patterns closest to it. Substituting equations **(D8)** into equation **(D7)**, it is possible to obtain the dual of the optimization problem, Scholkopf et. al. [94], wherein the primal variables have been eliminated to leave one equation in terms of the dual variables, $\{\alpha_i\}$, only. After some manipulation, it becomes, subject to the first condition in **(D8)**:

$$\text{maximise} \quad W(\alpha_i) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \bullet \mathbf{x}_j \right) \qquad \textbf{(D9)}$$

As a result of the second condition in **(D8)**, the hyperplane decision function can then be written as:

$$f(\mathbf{x}) = \text{sgn}\left(\sum_{i=1}^{n} \alpha_i y_i (\mathbf{x}^T \bullet \mathbf{x}_i) + b \right) \qquad \textbf{(D10)}$$

where b is computed from the Karush-Kuhn-Tucker (KKT), Bertsekas [8], complementary conditions $\alpha_i \left(y_i((\mathbf{w}^T \bullet \mathbf{x}_i) + b) - 1 \right) = 0 \quad i = 1,...n$.

It is worth noting that at this juncture, the one crucial property of both the quadratic programming problem and the hyperplane decision function, is that both of these functions depend only on the inner (scalar) product between patterns and it is this property that allows generalisation to non-linear cases.

D.4 Features spaces and kernels

Fig. D2 that follows depicts the basic concept behind the support vector machine:

- the training data in the input space is mapped via a non-linear mapping function Φ into a higher dimensional feature space then,
- a separating hyperplane with a maximum margin is constructed in this feature space (following the arguments of the previous section and fig. D1) and

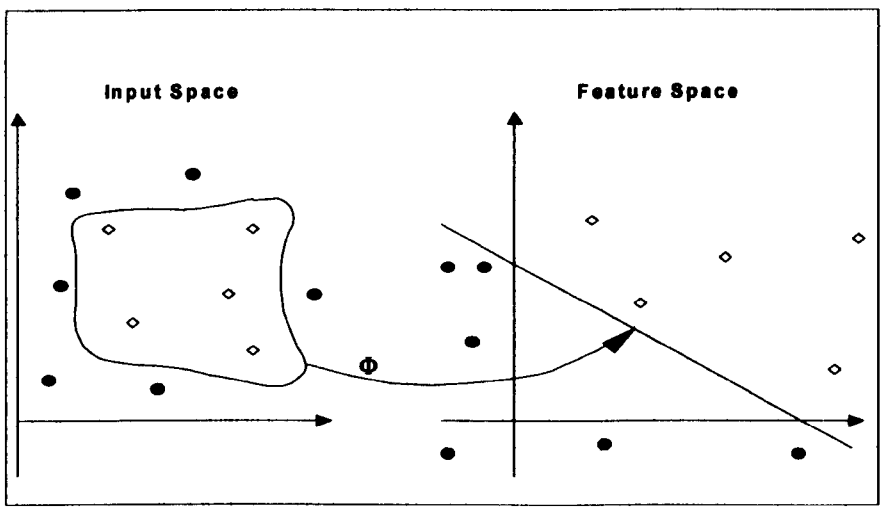

Figure D2: Input and feature spaces

- as a consequence, a non-linear decision boundary is obtained in the input space so that
- finally, using a kernel function, it is possible to compute this separating hyperplane without actually carrying out an explicit mapping into the feature space.

In essence then, the training data is mapped into some other inner product vector space, as mentioned above, called the feature space F using a non-linear mapping $\Phi : \mathbb{R}^n \rightarrow F$ and then a linear algorithm is applied in F. Unfortunately, if F is highly dimensional, evaluation of these products can be 'expensive' in terms of computer time and/or capacity. However, it can be seen that the construction of the optimal hyperplane in F, equation **(D9)** and the

corresponding decision function **(D10)** require just the evaluation of inner products such as $\Phi(\mathbf{x})\bullet\Phi(\mathbf{y})$ but not the mapped pattern $\Phi(\mathbf{x})$ in explicit form. This is a critical property since in certain cases, the inner products can be evaluated with a simple Mercer, [78] kernel such as $k(\mathbf{x},\mathbf{y}) = \Phi(\mathbf{x})\bullet\Phi(\mathbf{y})$. For example, the polynomial kernel k

$$k(\mathbf{x},\mathbf{y}) = \left(\mathbf{x}^T\bullet\mathbf{y}\right)^d \tag{D11}$$

corresponds to a mapping Φ where

$$\Phi:\left(\mathbf{x}^T\bullet\mathbf{y}\right)^d \rightarrow \Phi^T(\mathbf{x})\bullet\Phi(\mathbf{y}) \qquad \mathbf{x},\mathbf{y}\in\mathbb{R}^n \tag{D12}$$

In fact, if $d=2$ and $\mathbf{x},\mathbf{y}\in\mathbb{R}^2$, then

$$\begin{aligned} k(\mathbf{x},\mathbf{y}) &= \left(\mathbf{x}^T\bullet\mathbf{y}\right)^2 \\ &= \left(\begin{pmatrix}x_1\\x_2\end{pmatrix}^T\bullet\begin{pmatrix}y_1\\y_2\end{pmatrix}\right)^2 \\ &= \left(x_1y_1+x_2y_2\right)^2 \\ &= \left(x_1^2y_1^2+2x_1y_1x_2y_2+x_2^2y_2^2\right) \\ &\equiv \begin{pmatrix}x_1^2\\\sqrt{2}x_1x_2\\x_2^2\end{pmatrix}^T\bullet\begin{pmatrix}y_1^2\\\sqrt{2}y_1y_2\\y_2^2\end{pmatrix} \\ &= \Phi^T(\mathbf{x})\bullet\Phi(\mathbf{y}) \end{aligned} \tag{D13}$$

Scholkopf et. al. [94], point out that for every Mercer kernel that gives rise to a positive 'matrix' of form

$$\mathbf{K}_{ij} \equiv k(\mathbf{x}_i,\mathbf{y}_j) \;\; \forall\{\mathbf{x}_1\ldots\mathbf{x}_n\} \tag{D14}$$

then a non-linear mapping Φ can be constructed such that the kernel $k(\mathbf{x},\mathbf{y}) = \Phi(\mathbf{x})\bullet\Phi(\mathbf{y})$ is valid. Other kernel functions apart from **(D11)** are used in practice dependent upon the problem at hand, some examples of which are, respectively, RBF kernels and sigmoid kernels such as

$$k(\mathbf{x},\mathbf{y}) = e^{\left(\frac{-\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}\right)}, \quad \text{and} \quad k(\mathbf{x},\mathbf{y}) = \tanh(\kappa(\mathbf{x}^T\bullet\mathbf{y})+\theta)$$

where in the latter, κ is the gain and θ is the threshold.

D.5 Support vector machines

All the tools necessary to construct a non-linear classifier (as shown in fig. D2) such as Φ have now been discussed. In essence, the mapped result $\Phi(\mathbf{x}_i)$ is substituted for each of the $\{\mathbf{x}_i\}$ from the training set (in the input space) and then the optimal hyperplane algorithm (as indicated by fig. D1 and associated notes) is applied in the feature space. In the trivial example that was explored, equation **(D13)**, it would be necessary to first perform the substitution

$$(x_1 \quad x_2)_i \text{ with } \left(x_1^2 \quad \sqrt{2}x_1x_2 \quad x_2^2\right)_i$$

Since kernel functions and associated inner (scalar) products are being used, non-linear decision functions of form (cf. **(D10)**)

$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{n} \nu_i k(\mathbf{x}, \mathbf{x}_i) + b\right) \qquad \textbf{(D15)}$$

are realized where the parameters ν_i are obtained from the solution to the quadratic programming problem, for example: $\nu_i = y_i\alpha_i$ for pattern recognition, $\nu_i = \alpha_i^* - \alpha_i$ in regression estimation. It is noted that, in the input space, the hyperplane corresponds to a non-linear decision function, the form of which would be determined by the kernel function used.

So far, essentially only classification has been considered. However, Hearst et. al. [44], points out that it can be generalised to regression estimation where the objective is to estimate some $y \in \mathbb{R}$. Essentially then, an algorithm is used to construct a linear function in the feature space such that the training points lie within some neighbourhood ε, $\varepsilon \in \mathbb{R}^+$. Using similar arguments as for classifiers, this results in a quadratic programming problem in terms of the kernels, resulting in a non-linear regression estimate of form

$$f(\mathbf{x}) = \sum_{i=1}^{n} \nu_i k(\mathbf{x}, \mathbf{x}_i) + b \qquad \textbf{(D16)}$$

The similarity to **(D15)** is noted. To apply the algorithm, it is necessary to either specify ε *a priori* or fix an upper bound on the fraction of training points that are allowed to lie outside of the neighbourhood distance ε from the regression estimate.

D.6 Implementation and Development

Most usefully, it is it practical tool in that it reduces to a quadratic programming problem that has a unique solution. By a suitable choice of kernel functions, it is possible to construct different architectures. For example, RBF classifiers, polynomial classifiers and three layer neural networks could be represented respectively by:

$$k(\mathbf{x},\mathbf{y}) = e^{\left(\frac{-\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}\right)}, \quad k(\mathbf{x},\mathbf{y}) = \left(\mathbf{x}^T \bullet \mathbf{y}\right)^d \quad \text{and} \quad k(\mathbf{x},\mathbf{y}) = \tanh(\kappa(\mathbf{x}^T \bullet \mathbf{y}) + \theta)$$

Research is currently being conducted into better training methods for speeding up the solution to Q.P. problems and also into speed improvement in evaluation of the decision function, Burges [15]. Research is currently being pursued, Girosi [37], into the selection of suitable kernels and the associated feature spaces. Scholkopf et. al. [94], however, notes that the choice of kernels might not be so significant and they reservedly point out, that is does not imply that support vector methods will significantly outperform all other applications, or for that matter, solve a problem that has so far been intractable.

D.7 History of Support Vector Machines

The support vector (SV) algorithm is a non-linear generalization of an algorithm that was developed in Russia back in the sixties, Vapnik and Lerner [103]. A similar approach was taken by Mangasarian [72] and [73], in the USA. The development of the SVM gave rise to a class of algorithms, known as Kernel machines, for pattern recognition. The SVM was developed at AT&T Bell Laboratories by Vapnik and co-workers, Boser, Guyon and Vapnik. [11], Cortes and Vapnik, [25], being first introduced at a Conference on Learning Theory in 1992 (COLT-92).

Since their initial excellent performance as classifiers, SVMs have been shown to have similar performance in both regression, Schölkopf et. al. [94], as well as time series prediction applications, Muller et. al., [82], Drucker et. al., [34], Mattera and Haykin, [76]. Further, standard SVM toolboxes are also being developed, Cherkassky and Mulier, [21].

Appendix E
History of the Thames

E.1 Historical Development

The River Thames was initially formed between 170 and 140 million years ago as the result of the earth movements of the Jurassic Period. During the last glacial period, which ended about 18,500 BP when Great Britain was physically connected to Continental Europe, sea levels were much lower than they are today and the southern and central North Sea became land. At this time, the Thames was effectively a tributary of the River Rhine, the estuary of the latter being situated in what is referred to today as the Southern Bight of the North Sea. The Thames, the Rhine, the Medway as well as other rivers in the area fed a large freshwater lake situated in the Southern Bight. However, with temperatures slowly rising, the ice retreating and sea levels rising, the Thames became separated from the Rhine, the Straits of Dover became a marine channel and Britain became an island. Due to rising sea levels, the River Medway, instead of being an entirely freshwater tributary of the Thames, became a lower river source/estuarine tributary of the Thames.

E.2 Thames/Medway Physiography

From its now officially accepted source at Trewsbury Mead in the Cotswolds, the River Thames flows for approximately 67 Km through the southern counties to the Greater London area and on to Dartford Creek in the east before in essence, it can be regarded as estuarine. For approximately half of its length it is tidal in nature, the tidal influence starting at Teddington Lock (weir) 30.4 Km above London Bridge. If the River Medway is included as a tributary of the Thames, then the latter drains a total catchment area of some 12935.77 square Km.

The Thames estuary has the classic basic shape and characteristics. It is roughly bell shaped with freshwater entering it at the narrow upstream end (over Teddington weir in this case) and from any tributaries into it. Twice a day, the tides flow into the Thames as well as its tributaries. The volume of water varies with the amount of freshwater flowing into the Thames and the height of the tide. The physical characteristics of the tidal stream are very

different to the freshwater that flows over Teddington weir: it is saline, cooler (at least at the mouth of the river), quite well oxygenated, heavier and heavily silted.

In contrast to a freshwater river system wherein the flows are downstream all the time, in an estuary where tidal motion gives rise to water entering the system from the seaward end, the system becomes very complicated. As a consequence, the same water particles may flow back and forth for days and even weeks, before they reach the sea. Wheeler [108] notes that under certain conditions, centered around London Bridge, these particles may oscillate back and forth over a distance of between 12.9 and 14.5 Km. In the textbook situation where saltwater meets freshwater, the latter flows over the former as freshwater being less saline and hence not so dense, is lighter. However, where the two layers of water come into contact, mixing takes place with a gradual increase in the salinity of the freshwater.

However, the Thames does not conform to such a textbook scenario. It is in fact 'well mixed vertically'. That is, on average, the salinity declines gradually in the upstream direction. It has been suggested, Wheeler [108], that its nonconformity is caused by the series of bends in the river course between Teddington and Gravesend. There are in fact, 25 sharp bends between Teddington and the sea. Although the maximum tidal velocity is usually in midstream, at bends as the result of friction, it follows the outside curve of the river.

Consequently, the main tidal stream is continually switching from one side of the river to the other at such bends. This results in the freshwater and saltwater becoming thoroughly mixed so that at slack water, there is little difference between the salinity of the water near the bed and that at the surface. However, the overall salinity does gradually increase in a seaward direction. There are some small areas where little mixing has taken place for example at pronounced bends in the river where the tidal stream has been forced into eddies.

In addition to the freshwater flow over the Teddington weir, there are a number of tributaries, which flow into the tidal Thames that themselves are also tidal (referred to as tidal creeks). With two exceptions, they are not very large. These tributaries include the River Crane, the River Brent, the rivers Beverley Brook, Wandle, Ravensbourne and Quaggy from the south side, the rivers Lea, Roding, Beam and Ingrebourne from the north

side and the rivers Darent and Cray (combined at Dartford Creek) from the south side. The River Lea is without doubt the major tributary of the tidal Thames.

The freshwater flow over Teddington Weir is normally maintained at about 800 million litres/day although in the summer or severe drought conditions, it can be as low as 200 million litres/day. The mean flow (as measured over a 25 year period) is about 520 million litres/day. The difference in water level between Mean High Water and Mean Low Water (Mean Range) at London Bridge varies from 4.6 metres at Neap tides to 6.6 metres at Spring tides.

The River Medway has its source near Turners Hill, in West Sussex. It is tidal from Allington Lock to its confluence with the Thames, a distance of approximately 40 Km. The non-tidal length of the river to its source is approximately 50.5 Km (which includes Weir wood Reservoir of length 3.4 Km). The main tributaries of the Medway are: the rivers Eden, Bourne, Teise and Beuilt and the minor ones: Len, Loose and Shode. The headwaters of some of these tributaries are fed by springs that, in turn are fed by aquifers. The width of the Medway varies considerably with some sharp bends in its course. It is therefore a well mixed estuary. The Medway and the Eden rivers have a long history of flooding, especially the area of the upper Medway with its many tributaries. As a consequence, a flood barrier was constructed in the Tonbridge area.

Apart from the gauge station at Kingston (on the Thames) and that at Teston (on the Medway), all the other gauge stations that are used to estimate the freshwater flow into these major rivers, feed directly into them as there aren't any lock obstructions on route.

It can be seen from the CAD plan in appendix F that the bathymetry in the area surrounding the confluence of the two rivers is quite varied. In the Thames itself there are various sandbanks such as the *Great Nore* and *Nore Sand.* Other wetlands/sandbanks include *Chapman Sands*, *Southend Flat* and *Maplin Sands* on the northern side of the river and *Blyth Sands*, *Yantlet Flats*, *Grain Spit* and *Jacobs Bank* on the southern side. The main approach channel, that is dredged, is the *Yantlet Dredged Channel.* The Medway is approached via the *Medway Approach Channel* over a sandbank called the *Little Nore*

opposite *Garrison Point*. Proceeding up river leads to mud flats such as *Stoke Ooze* and to the south, an extensive area encompassing further mud flats and marsh lands such as *Ham Ooze*, *Bishop Ooze*, *Tailness Marshes* and *Chetney Marshes*.

Appendix F

Map of the confluence of the rivers Thames and Medway

Southend-On-Sea

Leigh-On-Sea

Benfleet Creek

Westcliff-On-Sea

Shoeburyness

Black Grounds

Thorpe Bay

Cambridge Town

Canvey Island

Marsh End Sand

Chapman Sands

Southend Flat

Maplin Sands

Hole Haven

Leigh Channel

Yantlet Dredged Channel

Blyth Sands

The Warp

Yantlet Flats

Nore Sand

Great Nore

Yantlet Creek

Grain Spit

Medway Approach Channel

The Cant

Isle of Grain

Little Nore

Cheyney Spit

Jacobs Bank

Garrison Pt.

Sheerness

Sheerness Harbour

Stoke Ooze

Saltpan Reach

Queenborough Spit

Queenborough Pt.

Burntwick Island

Slede Ooze

Kethole Reach

Tailness Marshes

Queenborough

Long Reach

Stangate Creek

West Swale

Rushenden Marshes

Bishop Ooze

Half Acre Creek

Ham Ooze

Greenborough Marshes

Chetney Marshes

Isle of Sheppey

Ferry Marshes

Legend

Below sea level

0

2

5

10

15

20

Scale = approx. 1/77500

Based on Admiralty Chart no. 1185 (scale 1/25000) 1997

All depth contours are in metric and reduced to chart datum.

Chart datum is approximately the level of the lowest astronomical tide (2.90m below OD).

L.H.Rees May 2006