# The real time visualization of artificial neural output during supervised learning

Senior Honours Project
14th of April 2000

Duncan McPherson

**University of St Andrews**

**Abstract**

This project report describes the design and implemenentation of two visualization tools which graphically show the changing state of an Artificial Neural Network (ANN) during supervised learning. JSAND - Java Simple Artificial Neural Display - allows the user to plot the input/output mapping of an ANN. An I/O mapping represents the complete set of output patterns of an ANN for all possible combinations of input pattern, according to the current weight state of that network. JSAND is able to render a three dimensional surface which accurately conveys the current I/O mapping of an ANN to the user as it changes dynamically in real time during supervised learning. Additionally JSAND is able to provide two dimensional I/O mapping plots in a variety of styles.

The second tool is able to plot a two dimensional path of the changing behaviour of an ANN in output space as supervised learning takes place. Both tools are able to be built as Java applet clients. In client form, these applets acquire ANN training data from a remote server which forwards data from a training program running on that server. Additionally the JSAND tool may also be built in Java application form where it allows basic ANN editing functions along with the ability to load and save neural networks to a local disc.

# Contents

# List of Figures

# Chapter 1

# Problem Definition

## 1.1 Problem Definition

There are two main objectives of this project. One is the generation of a real time three dimensional representation of the Input/Output (I/O) mapping of an Artificial Neural Network. An I/O mapping represents the complete set of output patterns of a neural network for all possible combinations of input pattern, according to the current weight state of that network. This project will allow a network's current I/O mapping to be rendered in real time while training of that network is underway.

The second objective is to allow the path of the output behaviour of the network to be rendered in output space. This representation is then to be extended to cater for the case when supervised learning takes place in stages, each new stage being a subgoal further toward the ideal output goal state.

### 1.1.1 Visualisation of input/output space

Typically the activation state of each neuron in a multi-layer neural network is calculated using a sigmoidal activation function. The complete input/output (I/O) mapping for such a network can then be found by feeding all possible combinations of input patterns into the network (up to a given resolution for real number inputs) to find each output pattern

in the mapping. This way of finding a networks I/O mapping is very time consuming and unsuitable for real time display of a networks current I/O mapping. Instead this project draws on the theory of hyperbands developed in St Andrews to approximate the sigmoidal activation function in real time. This will allow the production of a real time display of the networks I/O mapping.

The use of a semi-linear hyperband activation function allows the output responses over and area in input space to be quickly calculated by geometric means. This process can be repeated for all areas in the input space, and the results can be combined to give an overall I/O mapping for the current weight state of that network.

### 1.1.2 Visualisation of subgoal chains

The resulting output state of a neural network after training towards a goal output state is not always the best approximation of the goal output realisable. In many cases the training process fails to find the optimum set of weights the network should have in order that it may produce an output as close to the goal as possible. One solution to this is to simply keep re-starting the training until a good solution is found. Each training session starts with the network in a new random weight state, and over several training attempts a closer approximation to the goal output state may be found.

A different approach is taken for this project in an attempt to successfully train the network to an optimum output state in a single training session. Instead of training the network directly to the goal output state, a number of subgoal output states are chosen between the goal and the initial untrained output of the network. The aim is to place an optimal weight state close to the current weight state through placing a subgoal output state close to the current output state. With appropriately chosen sequences it is hoped that training of a network in this way will be much better at finding and keeping in an optimum output state that moves directly towards the goal output state rather than training directly.

The objective of this project is to be able to visualise the path that the network's changing output state follows under training and to extend this to visualise training with subgoals in

particular. This path is to be plotted in two dimensions, with a start point representing the initial (untrained) output state of the network and an end point of the ideal goal output state.

Since output states typically have more than two dimensions, a mathematical approximation of this path in n-dimensional output space is used here. Though information will be lost using this approach, the two dimensional output path should still be able to convey whether the training is successfully moving the networks output state towards the final goal output state or not. Additional plotting of a cubic spline curve between output states in two dimensions should allow output state transitions to be smoothly represented. Such a visualisation tool will also allow testing of hypotheses that only certain shapes of subgoal path or sequence are realisable in certain problems.

# Chapter 2

# Context Survey

## 2.1 Introduction to neural networks

Artificial Neural Networks (ANNs) are mathematical models, generally based on concepts learned from the study of animal nervous systems. Though they do not accurately model their biological counterparts, they still display several of the aptitudes that animal neural networks have evolved over millions of years.

ANNs are particularly useful in areas such as pattern recognition, where traditional statistical methods of computation have sometimes failed to produce good results. Their greatest strength is that given the same general network construction, they can be applied to a wide variety of different pattern recognition tasks. Whereas statistical analysis techniques used to interpret pattern based inputs would almost certainly require a new implementation for each specific problem.

Traditional AI approaches to pattern recognition also suffer from this failing. In order for an AI rule based search to yield good results for pattern comparison, in most cases it would first require hard coding of problem domain specific knowledge before being able to make useful comparisons. In any cases the neural network solution will avoid the need for this by being able to learn new patterns to recognise without the need for extensive redesign at the implementation level.

## 2.2 Network Structure

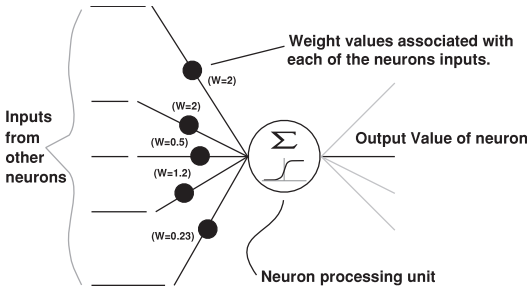


Figure 2.1: A neuron processing unit

For this project only feed forward, strictly layered neural networks will be considered. Given an appropriate number of layers a network can be trained to produce a number of desired outputs on recognition of an input pattern. Such a network is constructed from neuron units, several of which make up a single layer. Pattern recognition begins when the input layer of the network is presented with a pattern to process. Patterns can be analog or digital, with each input being given one value of the complete pattern. To this end, the network must have the same number of inputs as there are values forming the input pattern.

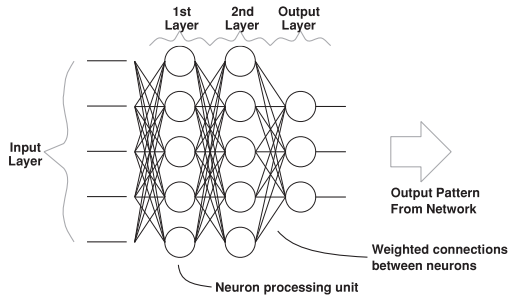


Figure 2.2: An example feed forward strictly layered neural network

Each input layer value is then passed forward to every neuron in the next layer. These forward connections each have an associated weight value which amplifies or attenuates the value being passed into the neuron. The neuron calculates an excitation value from all of its inputs and weights using the following formula :

$$\sum_i w_i x_i \qquad \begin{array}{rcl} x_i & = & \textit{Input value from previous neuron.} \\ w_i & = & \textit{Weight being applied.} \end{array}$$

Figure 2.3: A neuron's input excitation

From this excitation value, a neuron unit must decide its current output state. This is done by means of an activation function, in the simplest case a step function. When the excitation value computed exceeds a given threshold, the step function outputs a 1 otherwise it produces a 0.

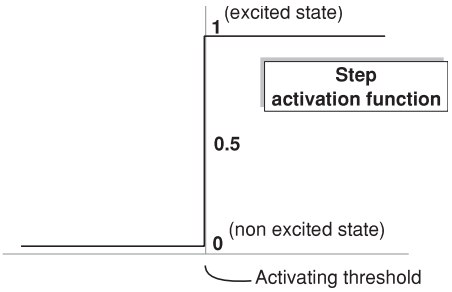


Figure 2.4: A step activation function

The weights associated with each onward connection are analog, and provide an arbitrary amplification or attenuation to the value being passed into the neuron. By this method, information being passed through the network is transformed by the weight values. All the networks knowledge is stored in these weights, and for the network to learn a new pattern association, training must take place to alter the networks weights.

## 2.3 Supervised Learning

When the network is untrained, its weight values should be set at random. Learning of a new input pattern can be accomplished by presenting the input neurons with the new pattern, then altering every weight by a general learning rule until the desired output

pattern is produced. This is known as a supervised learning strategy - where the correct output pattern is provided by the supervisor as a target to alter the weights toward. The weight alterations to meet this target are automated during the learning process.

If a step activation function is employed, only a single layer network may be successfully trained using the perceptron learning rule. For networks with hidden units further information is needed. Finding the amount to change each weight of a hidden layer neuron requires the computation of an error value for hidden units.

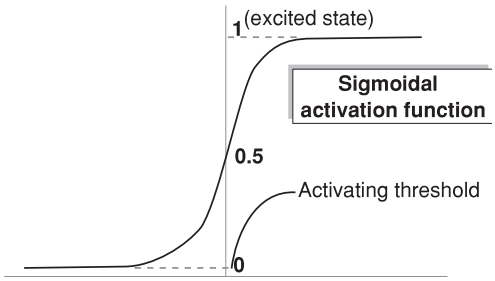


Figure 2.5: A sigmoidal activation function

A sigmoidal activation function allows such an error, called a delta error, to be computed from a derivative and used in conjunction with an appropriate learning rule. During supervised learning, the target output of the network and the networks actual output response to the training input can be computed. An error value for an output unit can be computed by subtracting the target output of that neuron from the actual output and squaring the result :

$$\text{Error value} = (\text{Target output of neuron - Actual output of neuron})^2$$

Back propagation learning specifies that the order of the weight changes will be last to first. That is, weights along links feeding into output neurons will be altered first, then those in the last hidden layer, and so on, with weights from inputs being changed last. In this way back propagation is able to pass delta errors backward through the network where each depends on the previous delta errors.

The weights associated with each neuron can then be altered using that neuron's delta error

value and a suitable learning rule. In order to learn successfully, the supervised output of the network should approximate the target output as closely as possible, that is, the value of the output neuron's error should be made as low as possible during training. Once this process is complete the network should have successfully have learned a new input output mapping based on the training set.

## 2.4 Subgoal chains

An ideal I/O mapping is that mapping associated with the global minimum produced by the goal weight state. This goal weight state is unknown a priori and must be searched for from the initial weight state of the network (see figure below). This search makes use of the known goal output state as a target to train the network towards. Travel to a local minimum during this search may produce a poor approximation of the goal weight state, so such local minima should be avoided by a good training strategy.

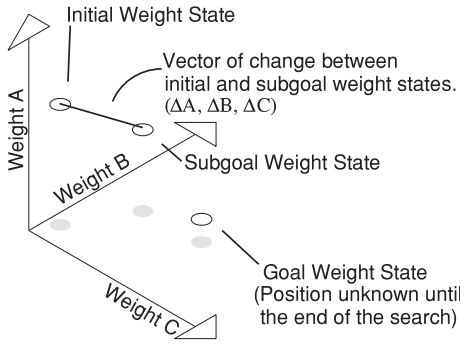


Figure 2.6: Changes in weight space during search for a goal weight state

Training the network to a number of subgoal output states before the goal output state is an attempt to avoid the problem of local minima preventing the global minimum goal weight state being found. A subgoal chain attempts to lead the search to the goal weight state by using nearby subgoals in output space in the hope that each subgoals global minima will also be nearby in weight space. The heuristic being used here is that a small change in output space should have a correspondingly small change in weight space. It is then hoped

that a search for a nearby subgoal in weight space is much more likely to succeed than a direct search for a faraway goal weight state.

## 2.5 Non-linear subgoal chains

A linear subgoal chain sets each subsequent subgoal to be proportionally nearer the final goal state than the previous subgoal. However it is known that linear subgoals can still become stuck in local minima under some circumstances.

The use of non-linear subgoal chains may be able to avoid local minima if each subgoal output state is positioned so that training avoids any local minima present in weight space. The testing of a new subgoal generation algorithms would benefit greatly from a display of the networks changing output state during training to each subgoal. This display should be able to indicate the networks progress or lack of toward each subgoal output state.

## 2.6 Visualisation of subgoal chains

Output space is the complete set of possible output states that a network may produce. In this respect it has the same number of dimensions as there are output neurons, and so a mathematical approximation of n-dimensional space must be produced for a two dimensional display. A vector calculated between two output states in output space will have both magnitude and an angle which can be calculated in an n-dimensional output space as follows:

Magnitude between two points where dx is a point in dimension x in a space with n dimensions :

Angle via Scalar Product between the two vectors a and b (using above formula to find magnitudes of each vector) :

The magnitude and angle calculated can then be represented as an output space point on a two dimensional plane. This allows the path of the networks learning through n-dimensional output space to be plotted in two dimensions.

$$\sqrt{\sum_{i=1}^{n} ( d_{i_1} - d_{i_2} )^2}$$

Figure 2.7:

$$\theta = \cos^{-1} \left( \frac{a \,.\, b}{|a||b|} \right)$$

Figure 2.8:

Though much information will be lost using this approach the two dimensional representation should produce an informative display of the networks learning path for different training problems. The most important information that it will convey is whether the new output space position after training to a subgoal is getting nearer the actual goal (and each subgoal), and whether it is heading directly for the goal, or not.

In order to visualise a network's progress towards a known goal output state in two dimensions, a reference vector between the networks initial (usually randomly set) output state and the goal output state should be found first. For the initial output space position, an arbitrary point on the two dimensional output space plane can be chosen. The reference vector between the networks initial output state and the goal output state can then be plotted with a magnitude calculated as described previously. For display purposes the goal output state reference vector will be plotted to the right of the initial output state, the appropriate magnitude away. All other output states to be displayed will then have their magnitude calculated from the initial output state, but will also require an angle to be calculated. This angle will be between the reference vector and the vector of the new output state position as detailed below :

In the case of linear subgoals where each subgoal is proportionally nearer the goal output

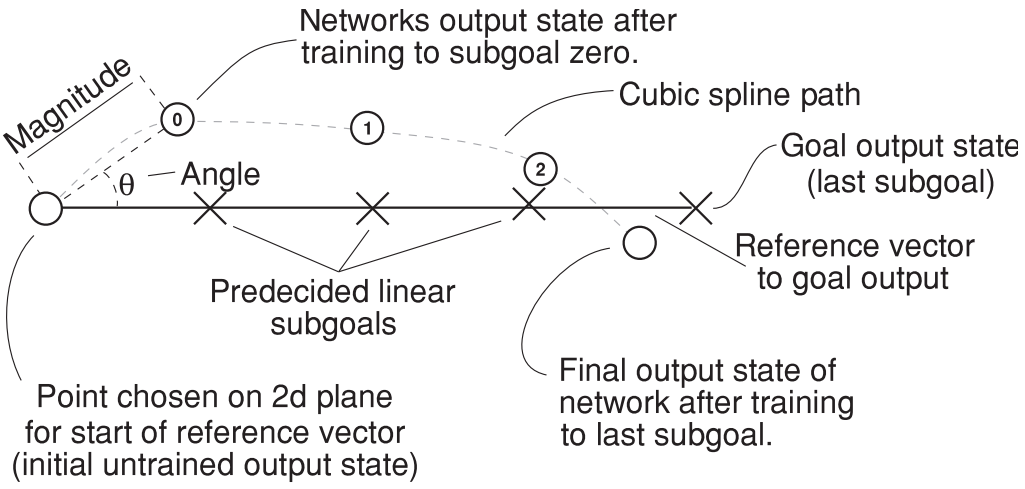


Figure 2.9: How the resulting 2d data will be plotted on a plane in the case of linear subgoals

state, all the subgoals will lie on the reference vector between the networks initial untrained output state and the goal state. In the case of non-linear subgoals, each subgoal output state will be determined by the given subgoal generation algorithm.

As the network is trained to each subgoal, the resulting output state can be calculated on the two dimensional output space plane. The collection of 2-d points will represent the networks path through output space as subgoal training is in progress.

A cubic spline curved path plotted between subsequent output states in two dimensions allows the output state changes being displayed to be smoothly represented. Such a curved output space path will also help to make trends in changing output state more immediately apparent to a user.

## 2.7 Input/Output Space

The concept of input/output mapping is a useful one during supervised learning, as it represents the complete set of output responses the network could produce for all possible input states and a given weight state. For a network with only one output and two inputs, I/O space can be represented in 3-dimensions, and the I/O mapping can be represented as

a 3-D graph. The number of hidden units in the network does not matter here, as only the input and output responses are being considered for the current weight state. As the weight state of the network changes, so too will the I/O space mapping. An I/O space display can then provide a useful indication of the extent to which the learning process has changed the network.

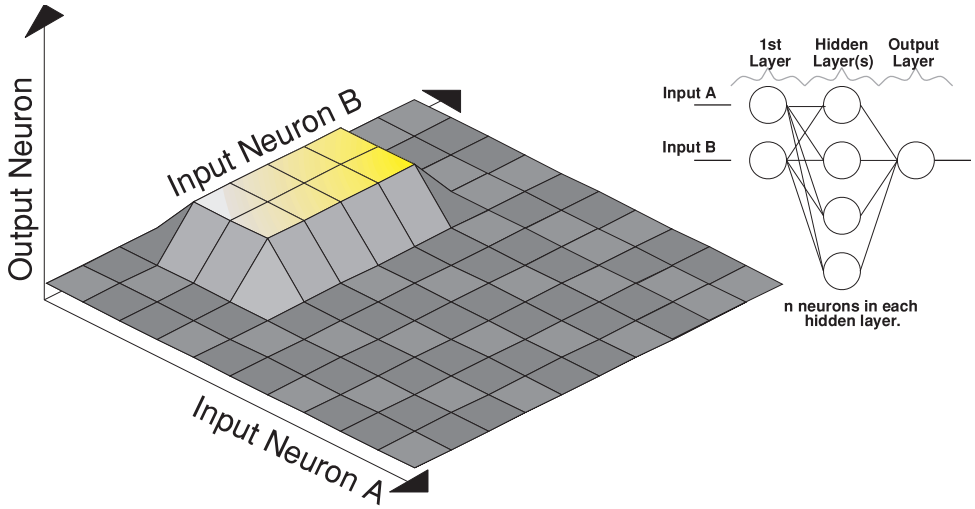


Figure 2.10: Low resolution (10*10) representation of an example network I/O space

Such a graphical representation of a networks I/O mapping can be generated by computing the networks response to every possible input permutation (to a given resolution for analogue inputs). This can be a slow process depending on the number of neuron units within the network, and the resolution of the display being produced.

## 2.8 Approximating I/O space using Hyperbands

For real time display of I/O space it is possible to use hyperbands to calculate a semi-linear approximation of the neural I/O mapping. A hyperband is the linear part of a semi-linear activation function (see figure below). This approach computes approximate activation characteristics for each neuron, and then combines the information to give a complete I/O space display.

A previous project ISLAND [4] took this approach to generate a real time network display, allowing the I/O space of a network with two floating point inputs to be displayed. Island produced a two dimensional display of the binary output of the network.

For this project, an analog display of the network output will be produced, rendered as an altitude on a three dimensional repesentation of a networks I/O mapping. Also networks used within ISLAND were required to only have one hidden layer of neurons, a limitation which will be partially alleviated in this implementation by allowing any portion of a network with a corresponding 2 - n - 1 topology to be used to generate an I/O space display. Networks with the topology 2 - n - m may also be catered for as an extension to the project by displaying several output space mappings, one for each of m output neurons.

Computing a neurons activity using a hyperband activation function allows a region to be determined within I/O space where that neurons activity characteristics are known. For a two dimensional input, the region is between the two parallel lines that form the hyperband.

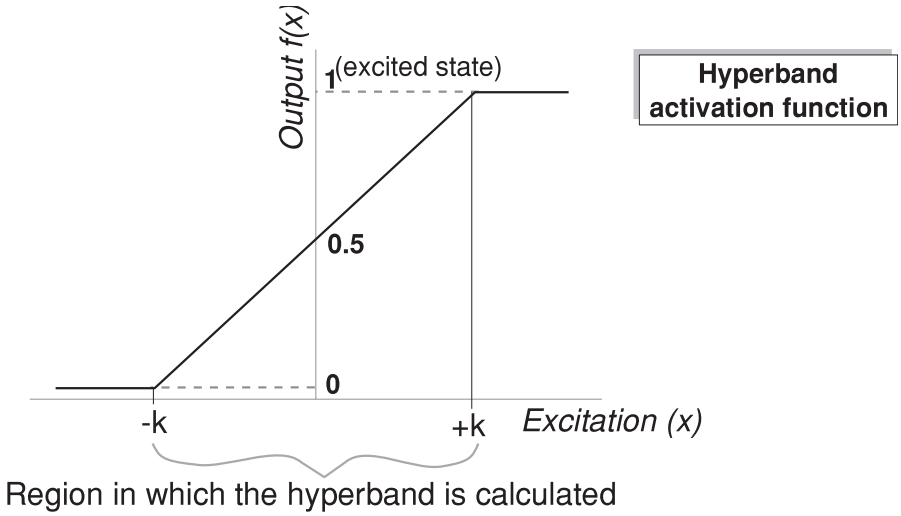


Figure 2.11: The semi-linear hyperband activation function

The semi-linear hyperband activation function above (f(x)) has three distinct output regions for a given neuron depending on the input excitation (x): The activity of a neuron as calculated by the hyperband activation function diagram (previous page) would be as follows:

1. f(x) = 0 where x < -k
2. f(x) = x/(2k)+0.5 where -k <= x <= k
3. f(x) = 1 where k < x

The hyperband associated with each non-input neuron can be geometrically represented as part of the I/O mapping in I/O space, as follows with a two dimensional input :

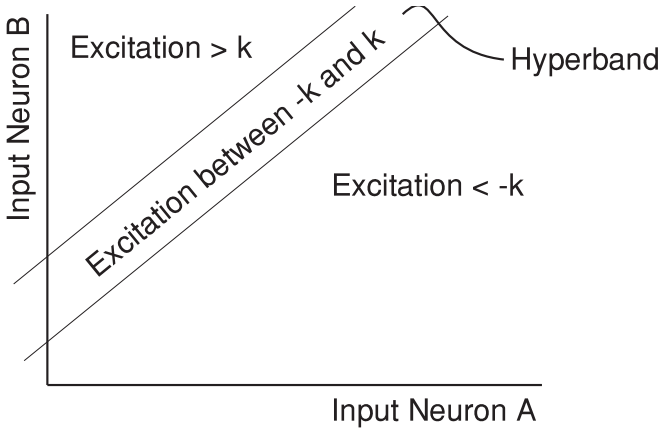


Figure 2.12: A Hyperband drawn in 2d I/O space

The lines bounding the hyperband correspond to the points at k and -k on the hyperband activation function. The values of k are chosen to ensure that the semi-linear hyperband function gives the best possible approximation of the linear sigmoidal function. The position and angle of these lines within output space can be calculated as a function of all a neurons weights, and present inputs as described by ISLAND [4].

Only binary output values can be determined using a hyperband. However for the purposes of this project, an analog display can be approximated by evaluating the neurons output characterisitics over a range of values in the linear part of the hyperband. Once all required hyperbands have been calculated, they can be combined to produce a display of the I/O mapping for the entire network.

# Chapter 3

# Problem Specification

## 3.1 Functional Requirements

### 3.1.1 Alteration of network properties

The software should provide a GUI based neural network structure and properties editor. This should allow changes to the topology of the network, and the values associated with each individual neuron. The network currently being modified within the editor will be available for direct use by both the I/O space display package, and subgoal chain display package, without the need to save the network.

Neural networks will be able to be saved as files, and loaded from file. Compatibility with other neural network file structures will be explored, hopefully allowing files created with Wintermute, Island, SkyNet packages to be imported and exported. Also to enable the use of an external training package, a specific neural network file will be able to be monitored, and reloaded if that file has been changed by another program.

### 3.1.2 Visualisation of the networks I/O space

The software should be able to render the I/O space for any given network with the general topology of 2 - n - 1, where n represents an arbitrary number of intermediate hidden layers

of neurons, each of which contain one or more neuron units.

I/O space will be displayed as a three dimensional altitude map projected onto a two dimensional window based display. Arbitrary resolutions of height map will be possible to represent the underlying I/O space mapping in greater detail. An upper bound for display resolution should be made according to available memory, and a maximum polygon quota giving acceptable system performance.

The display window will allow the three dimensional display of I/O space to be rotated in real time by the user, and both perspective and orthogonal projections will be possible.

In addition a region of a network may be chosen which corresponds to a 2 - n - 1 mapping and an I/O space display for this region only may be generated.

### 3.1.3 Visualisation of Subgoal chains

The visualisation of subgoal chains will be implemented via a two dimensional window based user interface. An initial point will be chosen on the left hand side of the window to represent the initial output state of the network. The reference vector between this initial output state and the goal output state will be transformed in two dimensions so that the goal output state lies on the right hand side of the window. The goal output state can then be plotted in the window, and all other output space points (and the path between them) will similarly be transformed in two dimensions using the same transformation. This will ensure that the two dimensional area where the networks output states are likely to be plotted during training is to be the central area of the window.

The calculations used to approximate n-dimensional output states to two dimensional display points are described in section 3.7. With training to each subsequent subgoal output state a new output space point will be plotted in the window.

In addition to plotting output space points, the path between them must be represented also as detailed in section 3.7 to help visualise trends in output state changes. The cubic spline curve being plotted will be determined using the two previous two dimensional output space points.

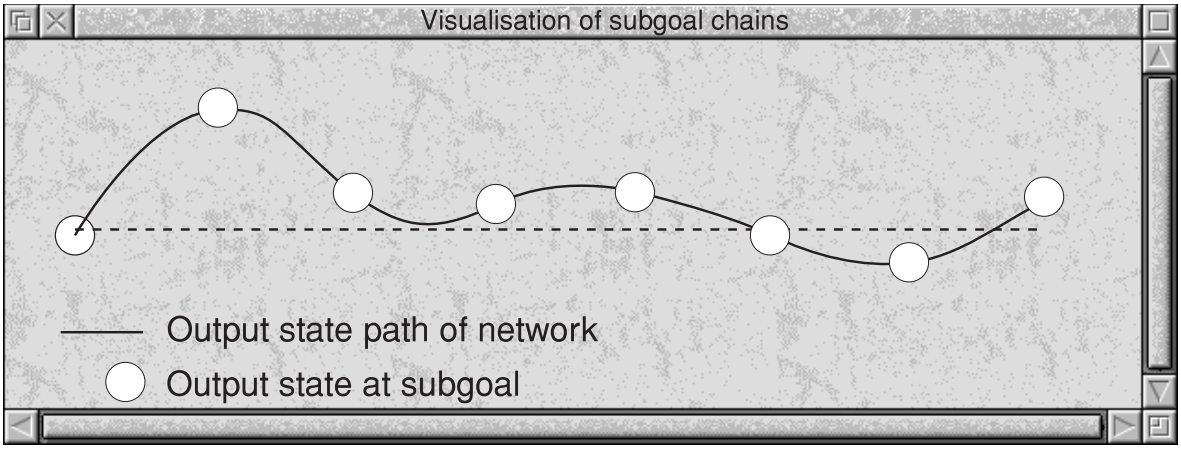


Figure 3.1: Example of subgoal chain visualisation based on a java user interface.

## 3.2 Non Functional Requirements

### 3.2.1 The use of host operating system libraries

The target platform for this project is to be a Solaris based PC computer as found in the honours laboratory. Several different software tools will be used to during the development of this software, in particular Sun Microsystems JDK - Java Development Toolkit, and also Sun's XGL graphics libraries for use within the Solaris environment. The main software engineering goals are to provide stable functionality, whilst gaining maximum portability of the software to future platforms, and maximum performance on the above specified hardware platform.

### 3.2.2 The user

The users is assumed to be familiar with the principles of neural networks, and also proficient at the use of graphical user interfaces. The software tools here are to be used within research situations and as such to be of any use, thorough understanding of background theory is required. In most cases this software will be used in conjunction with other network training software written by the user for a specific research purpose.

### 3.2.3 Documentation

Full maintenance documentation will be produced for the software for use in later development and alterations as necessary. This will describe all software modules produced, and their interactions. Detailed descriptions will be produced of the algorithms used, the core overall structure of each software module, and software interfaces between different modules.

User guide style documentation will be created with a complete and concise description of how to make use of the software, via the GUI provided.

Further documentation will be provided detailing additions to the core specification as described by this plan, and also any alterations to the plan that were made in hindsight, along with the design reasons for making them.

### 3.2.4 Performance constraints

In the case of three dimensional displays, the target performance is to be able to rotate these displays in 'real time' (i.e. where a frame rate of less than one frame a second is unacceptable performance). This should provide the user enough control over the display to adequately visualise the data being presented.

### 3.2.5 Error handling

Error conditions raised during the execution of the program should not cause the program to catastrophically fail (i.e terminate execution immediately, or halt the machine in a busy wait loop). Instead the user should be given notification that an error has occurred. If possible the user will also be given the option to continue the programs execution, or in the case of a recurrent or unrecoverable error, the user will be able to terminate the program.

If the user has chosen to terminate the program on an error condition, all open files will be closed, and control will be passed back to the host operating system in an orderly fashion.

### 3.2.6 Possible enhancements

Output to the ISO standard three dimensional modelling language VRML may also be possible with altitude maps generated for the visualisation of I/O space being converted to VRML indexed face sets within textual VRML output. This will allow a snapshot of I/O space to be exported to three dimensional modelling packages, and also to remote internet browser clients, equipped with a suitable VRML plugin.

In addition to testing the program on the target Solaris platform, it should also be possible to have the java only sections of the program functioning on other platforms, where only the subgoal chains package is available. Further enhancements could allow the java code to be compiled as an applet for use within an internet web page.

Visualisation of I/O space of networks with more than one hidden layer of neurons could be made possible with additional hyperband theory. In particular, visualisation of networks with two hidden layers would allow a much wider selection of neural network problem I/O spaces to be explored.

The I/O space of networks with the topology of 2 - n - m could also be made possible, having a separate I/O mapping displayed for each of the m output neurons. This would involve having multiple I/O space windows displayed simultaneously on screen, or a single I/O space window intelligently being split into an appropriate number of window 'panes' one for each I/O space mapping to be displayed.

The visualisation of subgoal chains currently only displays the path towards a goal output state within output space. It could be possible to also allow the n-dimensional path that the network is following within weight space to be plotted using the same theory. However since a goal weight state is not known, the trajectory of such a path would have to be interpreted without knowing the position it would be heading toward.

As this project is experimental, further useful display options may come to light during the implementation phase which could improve the users ability to visualise the data being presented. Any such additions to the project plan will be detailed in the final project report, and a special commentary made of the reasons for an addition.

# Chapter 4

# Commentary

## 4.1 Evolution of the project plan

The design and planning of this project to date has been under the supervision of Mike Wier (MKW) of the Division of Computer Science, St Andrews University.

Week 1 meeting - The project title and general overview were proposed by MKW. Reference was made to a previous honours project ISLAND (Storer 1994) which had a large theoretical overlap with this project.

Week 2 meeting - Initial theoretical groundwork concerning neural networks was discussed with MKW. Research references were given, along with an overview of the work of several related research efforts in particular ERA (Gorse et al 1995).

Week 3 meeting - Discussion of project theory continued, focusing on hyperband theory as developed at St Andrews University. A target was set to produce an initial implementation plan for one week hence.

Week 4 meeting - Initial implementation plan was reviewed, and a further target was set to produce a document detailing the neural network theory covered in previous meetings. There was also further discussion about subgoal chains, and possible ways in which they may be visualised.

Week 6 meeting - Corrections concerning the neural network theory document were reviewed, and several areas of confusion were cleared by MKW. Further targets were set for the production of a complete draft project plan.

Week 7 meeting - A mostly complete draft of the project plan was produced, and reviewed. During this process several technical misunderstandings were discussed.

Week 8 meeting - Final details of the project plan were reviewed, and a target was set to produce an overview section between the abstract problem definition, and the introduction to neural networks in the context survey. A draft of this was to be reviewed during week 8.

Week 8 final meeting - Several corrections to the overview section were discussed, and correction details of the draft project plan were finalised before submission at the end of week 8.

# Chapter 5

# Implementation and Testing Plans

## 5.1 Software resources to be used during development

### 5.1.1 The XGL library

The XGL graphics library will be used for the display of I/O space within the solaris X Windows GUI environment. This graphics library allows any graphics acceleration hardware available on the host workstation to be used. Also via the X windows network graphics model, XGL would allow viewing of the networks I/O space on a remote X terminal.

The Application Programmer Interface (API) for XGL is in the form of C libraries which are statically linked at compile time. These libraries provide for both 2d and 3d window based graphical displays. Detecting user commands to an XGL window is also possible so that interactions with the display can be implemented.

### 5.1.2 The use of the Java JVM and AWT

The Java language and associated virtual machine environment holds many advantages for the display a two dimensional subgoal chain representation, and also for the editing of a neural network.

The Java Virtual Machine (JVM) includes a full window manager, the Abstract Window Toolkit (AWT) which is accessible by the same software Application Programmer Interface (API) irrelevant to the actual host window manager. This allows software written for the AWT to be run on any hardware platform which has an associate JVM. In the case of Solaris, the AWT would make use of X Windows allowing the same network graphics model functionality as XGL (described previously).

Portability and reusability of Java source code is excellent, the former due to Sun Microsystems standard specification of the JVM which is closely followed by most JVM vendors. Reusability of Java source code is also easy relative to other languages such as C or C++, again due to the tight specification of the JVM and the feature rich Java OS libraries. Also the inherent object oriented structure of Java, removing the use of global variables, and forcing each software module defined as an object to communicate with other such objects via well defined channels.

### 5.1.3 Integrating Java and C software modules via the JNI

Integrating software components implemented in Java to machine specific object code is possible using the JNI - Java Native Interface. The JNI allows java classes to call native machine specific code and pass data to and from these native modules.

In the case of this project the main advantage gained through this approach is the use of the XGL rendering libraries available only to native code compiled for Solaris. These libraries surpass the primitive graphics functionality of Java in both speed and functionality, whilst losing the cross platform advantages inherent in Java.

To retain maximum portability of the project it is intended that whilst modules concerned with displaying I/O space are platform specific to Solaris, the remaining Java code will be able to compile and function independently, allowing the subgoal chain visualisation functionality to be useable on any JVM.

## 5.2 Software modules required

### 5.2.1 Summary descriptions of core modules

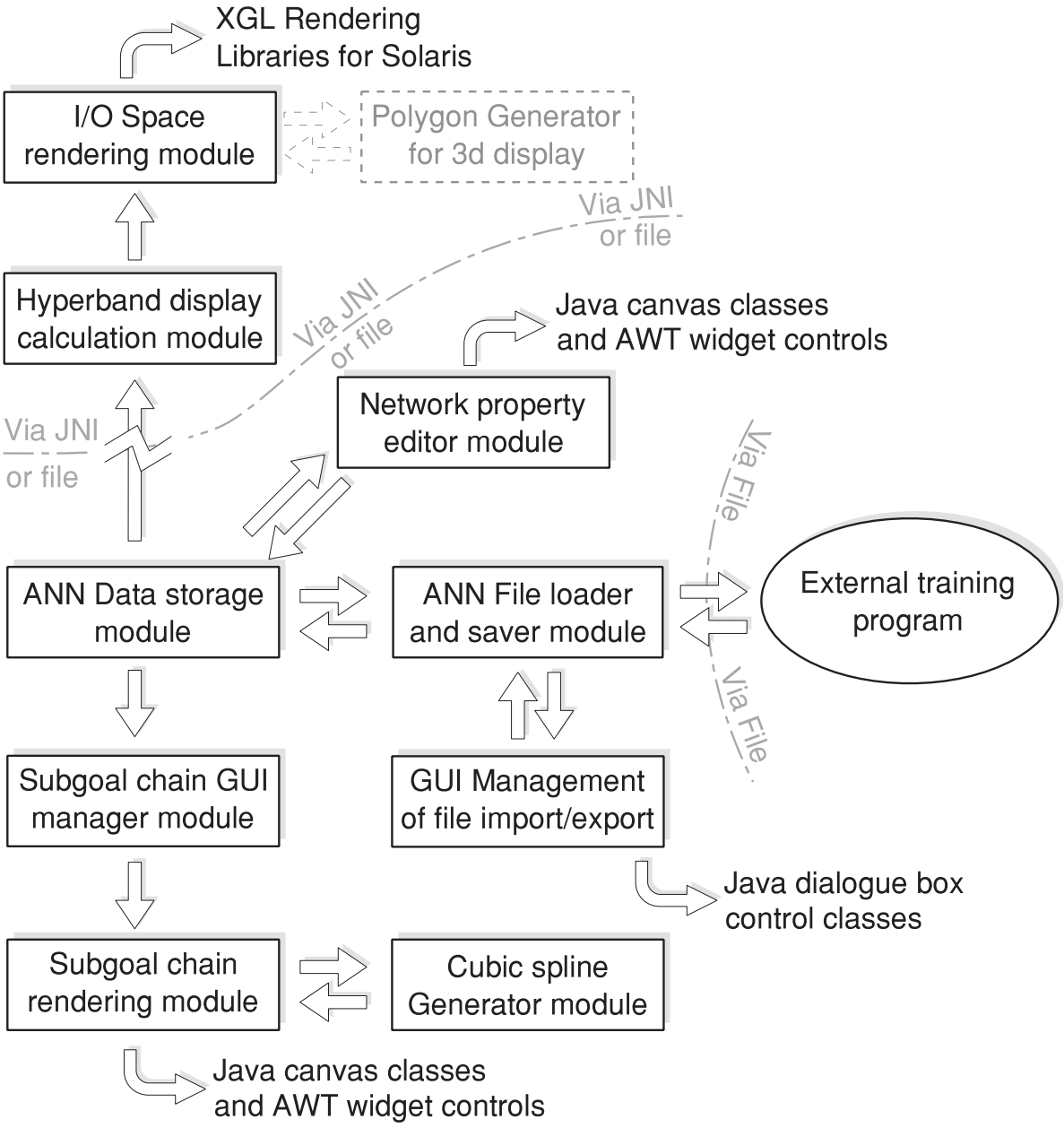


Figure 5.1: Overview of the software modules to be implemented as part of this project

The core modules within this project are those which are used by both subgoal chain visualisation, and I/O space rendering parts of the program. These core modules are to be developed in Java.

Network data storage module - This software module stores the currently loaded neural network within memory. It contains java methods used to compute I/O characteristics of the network for use by the subgoal chain display modules, and the I/O space rendering modules. Appropriate JNI header files to allow these methods to be accessed from C will be generated for this module.

Network property editor module - This modules allows the properties of the currently loaded network to be altered, or a new network to be created by the user. All network weights will be alterable by an appropriate dialogue box, along with associated bias properties for each neuron. An intuitive slider bar typed interface should be made use of here, allowing changes to be made to values easily using a mouse. Values should also be able to be manually entered at the keyboard.

ANN File loader and saver module - This module loads a specified file, and given that the file has an appropriate neural network file structure, it will be parsed, and the new network data passed into the network data storage module to become the current network. It should also be able to save the current neural network data for later re-loading.

GUI Management of file import and export - The module provides dialogue box functionality allowing a file to be chosen from disc and loaded into memory. A save as dialogue box will also be provided to allow a pathname to be chosen for the current network to be saved to.

### 5.2.2 Overview of I/O space display modules

The I/O space display modules are to be implemented in C, with communication between other core Java based modules via the JNI.

Hyperband display calculation module - All geometric calculations used to determine the location of the hyperbands will be carried out by this C software module. All input network data will be taken from the network data storage module and output will be in the form of

a two dimensional floating point array. This array will contain data displayable as either a colour representation of the network I/O mapping, or projected as a three dimensional altitude map of the network I/O mapping.

The display module will also receive control information from the network storage module informing it that the array should be recalculated, as the network weight state has just been changed.

If a specific part of the network is being targeted for display (a subsection of the network which has the topology 2-n-1) then information will also be passed to the hyperband display module informing it which part of the network is to be displayed.

I/O Space rendering module - This renders the contents of the array to an XGL window. The three dimensional display module is used to generate appropriately coloured polygon data for final output to the screen. All output is done via XGL library calls.

Polygon generator for 3d display - This module, given the complete 2d array of current I/O mapping will generate a database of 3d polygons for later rendering via XGL. The decision as to whether this database should be regenerated for each repaint of the XGL window, or only when the network data is to be changed will be taken in the light of performance data taken from a working implementation.

### 5.2.3 Overview of subgoal chain display modules

Subgoal chain GUI manager module - This module handles the storage of previous subgoal positions on the 2d subgoal display, and also calculation of the networks current output space subgoal position via class methods provided in the network data storage module.

Once calculated these position data points are also supplied to the cubic spline generator module, which generates an array of points which at the resolution of the output display will appear to be a smooth cubic spline curve. This curve data along with the output space points will be supplied to the subgoal chain rendering module which handles output to an AWT window.

Cubic spline generation module - This module, given the current list of output space points to be plotted to the window canvas, generates a further array of points which may be used to plot a smooth cubic spline curve. The number of points generated by this module will be dependent on the current display resolution, with a maximum of one point per vertical pixel of the display window canvas.

Subgoal chain GUI canvas module - This module handles the display of the subgoal chain window, and plots the points and associated cubic spline curve, as required by the AWT. It also handles user interaction with this window in order to change display options associated with subgoal chains.

## 5.3 Integration and testing of the modules

Each modules's functionality will be tested individually before the modules are integrated together to form the final package. Testing will be done to ensure that error conditions are appropriately handled. This will involve giving extreme input data conditions and invalid data conditions to a module and ensuring that it's response is appropriate as detailed in the problem specification. Modules will be tested individually with as many valid data sets as development time permits.

In C based source code macros will be used to assert conditions such as maximum array boundaries, and valid parameter ranges. Use will be made of preprocessor directives (e.g. the `LINE` and `FILE` directives) in order to trace faults within modules. Java based code will make use of hand coded assertions to help find coding errors.

Integration of the modules will require C and Java modules to be linked together using utilities provided by the JDK. Particular testing focus will be made upon the JNI data communications bridge between the Java and C modules. Testing will be done to ensure that data structures and methods are being accessed correctly between modules.

Finally, once successful integration of all modules has taken place, the program will be soak tested with as many network configurations as development time permits.

## 5.4 Deliverables

This project will be formally documented in two parts, the plan and the final report. In addition these documents, the software developed will be demonstrated on a date specified by St Andrews University Computer Science Division.

Project Plan - Due for completion on Friday the 13th of November 1998 at 4.00 pm.

- Contents
- Problem Definition
- Context survey
- Problem specification
- Commentary
- Implementation and Testing plans
- Project Monitoring Sheet
- References.

Project Report - Due for completion on Friday the 16th of April 1999 at 4.00 pm.

- Abstract
- Contents
- Project Plan (As described above)
- Project details
- Critical appraisal
- Testing summary
- User manual

- Maintenance document
- Software listings
- Status report

# Chapter 6

# Revised Project Plan Timetable

At the end of meeting 1 on the 4th February 2000, between myself and Dr Mike Weir, a set of proposed changes for this Senior Honours (SH) project were under consideration, in the light of new research work undertaken by Dr Weir's group between March 1999 and February 2000.

As initially conceived, this SH project would involve the development of two distinct functional components. A development plan covering both components was accepted by the Department of Computer Science on November the 13th 1998. The first of the two components was intended to allow the visualization of supervised learning in an Artificial Neural Network (ANN) using subgoal chains. The software deliverable for this part of the project was to render a two dimensional display of each subgoal to be used in the supervised learning session and then be able to superimpose a plot of the actual learning achievement of the network toward each of these subgoals, as learning progressed.

By March 1999 work towards an initial implementation of this subgoal component was well underway and a partially functional demonstration version of the subgoal renderer was submitted to Dr Weir at that time. This demonstration software has subsequently been maintained and enhanced by members of Dr Weir's group, as an aid in their own research with all changes having been made available to me in February 2000.

The second part of this SH project was to develop a two/three dimensional display of the

current input/output mapping for a given two input ANN. Changes have been made to the original planned implementation of this part of the project - primarily to allow the entire input/output display system to be written in Java. The advantages in this approach are described in the following proposed changes section.

## 6.1 Specification changes proposed during February 2000

In the light of changes made to the subgoal component software, and discussions of possible new enhancements to the software as it stands, the following development work is proposed:

- Enhancement of the current subgoal display system to allow the use of selected pairs of neural outputs. This is in contrast to the demonstration version of March 1999, which would only generate displays from a function of all the current neural output states for subgoals and also the current state of the network,
- Better integration of the display system with the ANN network simulator program provided by Dr Weir's group, so that the display is generated in real time from the output weight states provided by the simulator. The March 1999 version of the display system requires recompilation in order to display a different set of weight states.
- Altering the subgoal chain display system in any way that is necessary to allow seamless integration with the zip model of subgoal learning that has been developed by Dr Weir's group over the past 11 months.
- Evaluate the resulting system for the potential to be re-implemented in the form of a web applet program. This would allow subgoal chain displays to be made available to users as part of a web page in a Java enabled web browser. This aspect of the project would depend on portability of the network simulator - written in C - , which would have to be able to supply the display applet weight values. Initial thoughts are that a client server situation would be suitable with the simulator running on a server machine, and supplying a client side display Java applet with weight state data via a Berkeley socket connection. This form of connection between an applet and

its originator connection is permitted under the Java security model, provided the users web browser has not been configured to prohibit applets making any Internet connections.

- Development of the input/output mapping display component has been decided to continue with the immediate re-implementation in Java of parts of Storers original C code implementation of ISLAND. Once a complete Java implementation is available this component of the SH project can also be evaluated for use as a web applet and any changes necessary to achieve this end made.

## 6.2 Revised software development plan

The proposed changes section of this chapter and the following software development timetable was submitted to Dr Weir on February 4th 2000. From there the seven week schedule was followed reasonably closely, though software development work continued up to and including week eight. In the light of implementation experience the XGL graphics libraries have not been used for 3d rendering. Instead as partially described in the possible enhancements section of the original plan all display code has been written in Java and the finished software systems for both subgoal chain and input/output space display are able to run as Java applets in Internet web pages. Additionally instead of using the JNI as a communications bridge between the external training program and the display applet, a client server system has been implemented to transfer training data from a training program running remotely to the client applet. This has the advantage of making the display client applets completely platform independent, as they interact with the remote server program to receive updates. A C based server program has been implemented to allow this functionality. Finally the completed input/output space display system may also be built in the form of a Java application. In addition to the functionality provided by the applet build the application build also allows neural networks to be filed on disc and edited manually. The revised development timetable proposed was as follows :

Week 2, 7-11th February

Discuss this timetable with mkw. Possible meeting with mkw's research group members to receive source code for ISLAND and their altered version of the subgoal display component. Initial developments in Java using Storers original ISLAND source. Alteration of subgoal chain program to allow selection of inputs. Initial changes and development to be submitted to mkw on Tuesday 15th February.

Week 3, 14-18th February

Feedback on initial changes to subgoal code from mkw. Discussion of Hyperband theory, and enhancements of Storers ISLAND system. Implementation of communications bridge between subgoal chain display and network simulator (Java Native Interface / Sockets connection). Completed implementation - and rudimentary documentation - for all neural network data structures as used for Island. Submission of this to mkw on Tuesday 22nd February

Week 4, 21-25th February

Work towards an initial submission in Java of a working version of Storer's ISLAND system. Possibly at this stage omitting any theoretical enhancements discussed during week 3 however with strong focus on having the display GUI parts of this system operating as a web page applet. Tentative submission date Monday 28th February. Possible further discussions with mkw on the enhancements of Storers ISLAND system for 3d/2d colour displays.

Week 5, 28-3rd March

Continued work on Storers system, and submission of a working subgoal display system, with communications bridge operational so that live data may be accepted from the network simulator without the need for recompilation. Hopefully also the subgoal display system will work as a Java applet. Submission date Monday the 6th March.

Week 6, 6-10th March

Discussion of the submitted subgoal system, and continued refinement of the Java ISLAND

system. Discussion of required parts for the SH project write up with a view to submission of a draft write up - complete or only core sections - during week 7.

Week 7, 13-17th March

Demonstration of the Java ISLAND including all enhancements either 2d or 3d output. Also demonstration of the subgoal system as a Java applet, or in the event of technical difficulties as a Java application running in concert with the simulator. Demonstration will take into account initial feedback from mkw's group members in the light of the submission of Monday 6th of March.

Easter Vacation

Consolidate all documentation produced so far into a final SH project report, created in LaTeX.

Week 8, 10-14th April

Final revisions to both the software and write-up of this SH project.

Week 9, 17-21st April

Report due Friday 14th of April, with acceptance testing beginning Monday the 17th April.

# Chapter 7

# Project Details

This project provides two separate display systems each of which allows the current state of a neural network to be visualised dynamically during supervised learning. Both display systems share some common code in their applet form, as they both rely on the same server system to supply them with training data to render. The overview of software modules in the implementation section of the plan indicates this overlap in functionality as modules common to both systems are only shown once. However apart from using the same server system to source data both display systems are completely separate, and will be described here as such.

## 7.1 The JSAND input output space display

JSAND - Java Simple Artificial Neural Display - the input/output (IO) space display system allows the user to supply a feedforward single layer neural network constrained to a 2-n-1 topology. When a neural network weight state has been loaded into JSAND the complete IO mapping of that network will be displayed for that networks current weight state. The user has a choice of three possible display modes, *Island* : two dimensional, *Wintermute* : two dimensional and *Jsand* : three dimensional. Figure 7.1 illustrates the same neural network IO space for a network with two hidden units (n=2) in Island mode (left) Wintermute mode

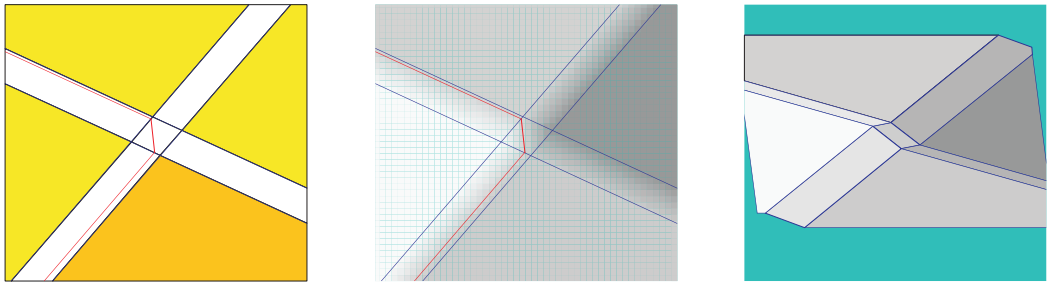

Figure 7.1: The same IO space viewed from the three JSAND display modes

(center) and Jsand mode (right).

The first two modes - Wintermute and Island - are so named because of their similarity to previously available IO space display systems. As a two dimensional display, each pixel at integer input coordinates (X,Y) on the display plane should represent a possible output activation of the ANN. The pixels across the X dimension of the display are used to represent the range of possible input values for the first input neuron, and the Y dimension the second input.

### 7.1.1 The Wintermute display mode

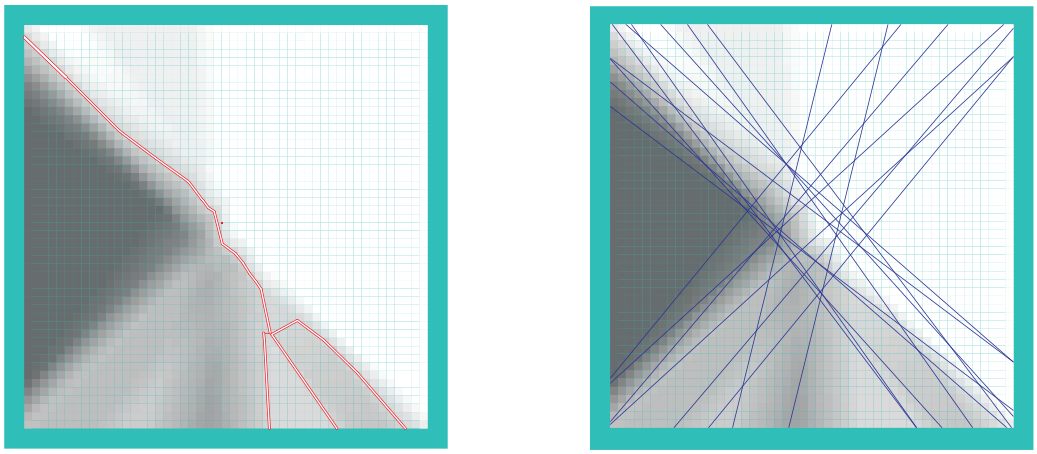

Figure 7.2: The Wintermute display mode with ten hyperbands

The Wintermute display mode splits the on screen display area into a grid of smaller square

areas. The activation of the current network is then found for each area in the grid, and the areas are then coloured a shade of grey corresponding to their activation - a real number between zero and one with black for zero and white for one. Generation of an IO space display by the Wintermute method is slow however, as each square area must be considered individually and its x and y input coordinates fed into the network to compute the output activation (see figure 7.2).

### 7.1.2 The Island display mode

In Island mode the mathematical characteristics of a semi-linear activation function are made use of to calculate the two dimensional IO space display. Hyperbands, areas in IO space in which the excitation of a hidden layer neuron varies linearly between +- the constant K are plotted, with one hyperband per hidden layer neuron. The whole rectangular display area is then split into regions, where the edges of hyperbands intersect. Hyperbands are represented as two parallel lines in IO space, both of which may be represented by the straight line formula y = mx + c.

Splitting the display into regions begins with a single region, the four sided rectangular display area itself. This region is then split into parts wherever the lines of a hyperband intersect it. The splitting process continues until no hyperband lines intersect any of the resulting regions. Figure 7.3 shows how a single rectangular display region is split into eighteen regions wherever hyperbands intersect. The right hand side of the figure shows the resulting regions in an exploded view where regions have been spaced apart from each other to better illustrate the result of the splitting process.

Once all regions are known, the behaviour of the output neuron within that region may be calculated. From the mathematical analysis provided by Storer [4] a further straight line formula for each region may be computed. This new line is known as the boundary line, and represents the line at which the network's output neuron will have an excitation of zero and activation of 0.5. Boundary lines are calculated with respect to each region, and are only valid if they intersect the region for which they were calculated. Any boundary lines falling outwith their region are discarded from the display. Some of the regions shown in the

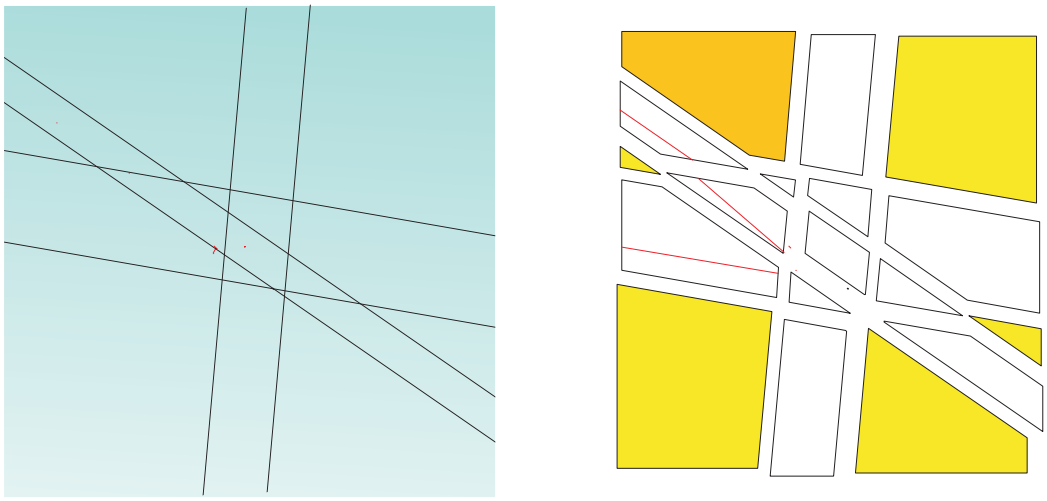

Figure 7.3: Splitting the display into regions

right hand half of figure 7.3 have boundary lines through them - lines which are contained entirely within the separated convex region shapes.

After computation all boundary lines that were found within regions may be plotted. In Storer's original implementation of ISLAND these boundary lines are then used to further split regions into subregions. Each subregion was then coloured by feeding the center (X,Y) coordinates through the network and finding a resulting output activation. Since all subregions generated using this algorithm were convex, the center of a subregions was easily calculated by averaging X and Y coordinates from each point defining that subregion.

The JSAND Island mode display only shows boundary lines, no colouring with respect to the output activation of network is performed. In Island display mode JSAND colours all regions falling within hyperbands white, and regions outwith hyperbands either yellow or orange. Yellow regions lie on the 'on' side of one or more hyperbands in the display, and orange regions fall on the off side of all hyperbands in the display. On and off values are calculated with respect to the output activation of the hidden neuron from which a hyperband is calculated.

JSAND does however allow users to superimpose red boundary lines over a Wintermute mode display. The left hand side of figure 7.2 shows boundary lines clearly separating

IO space where the network output activation is above and below 0.5. The boundary lines shown in figure 7.2 have been enhanced for clarity with a thinner copy of themselves superimposed in white - studying the figure shows how the boundary lines surround the light coloured areas of IO space where the network activation is above 0.5. This allows users to quickly see the analogue output on either sides of the boundary and intuitively know by the greyness of the regions which side is above and which is below 0.5. In this way the Island and Wintermute display modes are complimentary. GUI controls are provided allowing users to toggle the display of hyperbands and boundary lines in either mode.

### 7.1.3 The Jsand display mode

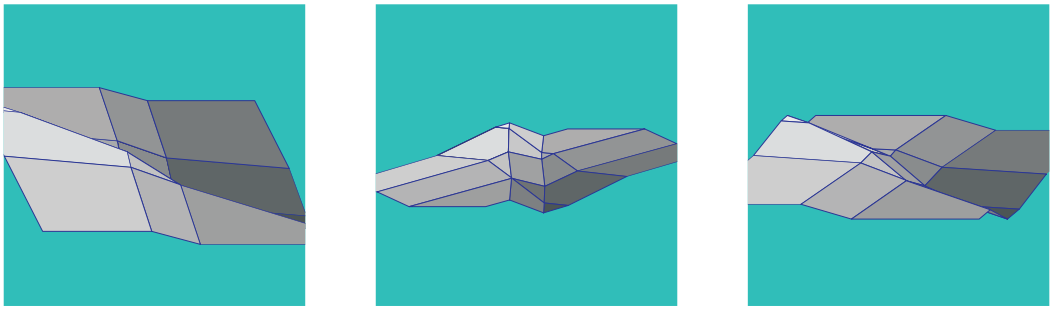

Figure 7.4: The same IO space display in 3d from 3 different viewpoints

The unique Jsand three dimensional display mode affords the user a highly intuitive way to visualise the current IO mapping of a neural network. In this mode, regions are calculated as with Island mode, and then given a grey scale colouring by finding the activation of their centers. From there each (X,Y) point comprising a region is converted to an (X,Y,Z) three dimensional vertex. The X and Z coordinates are the original X and Y respectively. The new Y coordinate in XYZ is the activation of the network at the original XY position in IO space multiplied by an altitude scale factor. All the regions in the display are then transformed by rotating their vertices about the y and x axes so that the IO space display is rotated to the current user selected 3d rotation angle, and then plotted onto the screen. In the Jsand display mode the user is free to change the angle of rotation by dragging the mouse across the display area. As the mouse is dragged, the rotation angles about x and y

axes are altered by the distance the mouse has been dragged allowing the user to rotate the display in real time (see figure 7.4). The origins of both the 3d IO space display and the 2d screen display are always at the center of the users on screen JSAND window, ensuring rotation is always about the visual center of that window.

The Jsand display mode allows for both wire frame and solid plotting. When the 3d regions are plotted solidly, the order of plotting becomes important. Those regions that are furthest away in Z - the axis perpendicular to the users screen - must be plotted first so that nearer regions will be plotted on top of them. To achieve this type of hidden surface removal each 3d region is given a depth index which is equal to the furthest Z coordinate of any of its vertices. All regions are then sorted in order of increasing depth index. This provides a solid orthogonal projection of IO space in three dimensions. As the Jsand display is orthogonal and not perspective corrected, Jsand is able to rotate the display so that it is viewed from above when the Jsand display mode is first selected. The overhead view of IO space closely resembles the Wintermute display mode, and from there users can rotate the IO space surface at will to better understand its shape.

As activation within each 3d region behaves linearly the altitude mapped Jsand display accurately reflects the IO space mapping of an ANN when a hyperband activation function is being used.

### 7.1.4 The load and save functions of the JSAND Application

The JSAND program makes use of a file format specified jointly by myself and members of Dr Weir's research group at St Andrews. For backward compatibility it is closely related to the file format used by the ISLAND program, and so very little conversion work should be needed to allow ISLAND files to be imported into JSAND. Descriptions of a neural network are saved as ASCII text, with specific tokens being associated with one or more integer and floating point values. The textual tokens used are as follows:

- `_Neuron %i` - Indicates that a description for a new neuron unit is to follow. The integer parameter %i is the neuron number.

- `Type %i` - Indicates the type of neuron. The integer parameter types are 1 for an input neuron, 3 for a hidden neuron, 2 for an output neuron and 7 for a bias neuron.
- `From Unit: %i Weight: %f` - Indicates a weight value joining the current neuron unit to another neuron unit number %i. %f denotes the real number value of the weight.
- `To Unit: %i Weight: %f` - Indicates a weight value joining two neurons.
- `Done` - This token is used to indicate the end of a complete neural network description within the file. When discovered within a stream the parsar should cause the display to be updated with the new network description.

Both the parser module and the ANN data storage module of JSAND allow neural networks not restricted to a 2-n-1 topology to be loaded and manipulated. However because the JSAND IO space display is 2-n-1 specific the parser will reject neural network weight state description files which do not describe a 2-n-1 network. If a network description file contains descriptions for more than one complete ANN (separated by `Done` tokens) each separate description will be loaded in sequence to form an animation with an 0.5 second gap between frames. In this way the complete saved output from an ANN training program can be loaded into JSAND and the changing IO space display viewed as an animation in either Jsand, Wintermute or Island modes. All of the usual display options are available, and may even be changed dynamically during animation - for example the current viewing angle or toggeling the display of boundary lines. This flexibility is allowed for because the JSAND file loader module is executed as a separate Java thread to the main JSAND application thread. The loader thread causes the display to be updated when each frame of the animation has been successfully loaded.

### 7.1.5 The JSAND applet in client mode

When JSAND is used in the applet client server configuration, all textual output from the C based training program is forwarded to the JSAND applet via a BSD socket connection. At

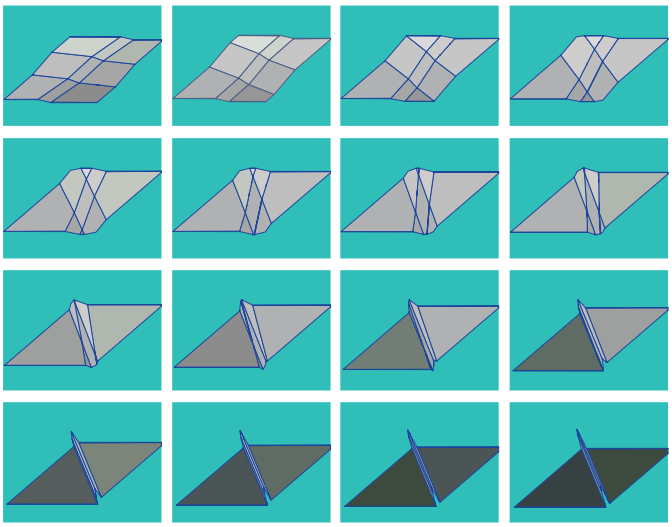

Figure 7.5: Animation of changing IO space during supervised learning

the server side of this system, a small C based server has been written which awaits socket connections from remote JSAND applets. On receiving a connection request from an applet, the server replies with an acknowledgement, and then initialises the neural network training program. If initialisation was successful all further textual output from the training program is then forwarded to the applet. The C based server runs under Solaris, an operating environment allowing the output stream from the training program to be easily attached to a socket connection.

Once a connection has been established and the JSAND applet has begun receiving textual data from the training program, the applet begins parsing this data stream. The neural network state information conveyed in the stream is in exactly the same format as JSAND application files, with any extra information in the stream being ignored. Once a complete new weight state for the current network has been received, and a `Done` token detected in the stream the JSAND applet updates the display. An example cartoon animation sequence of a neural network IO space during supervised learning is shown in figure 7.5.

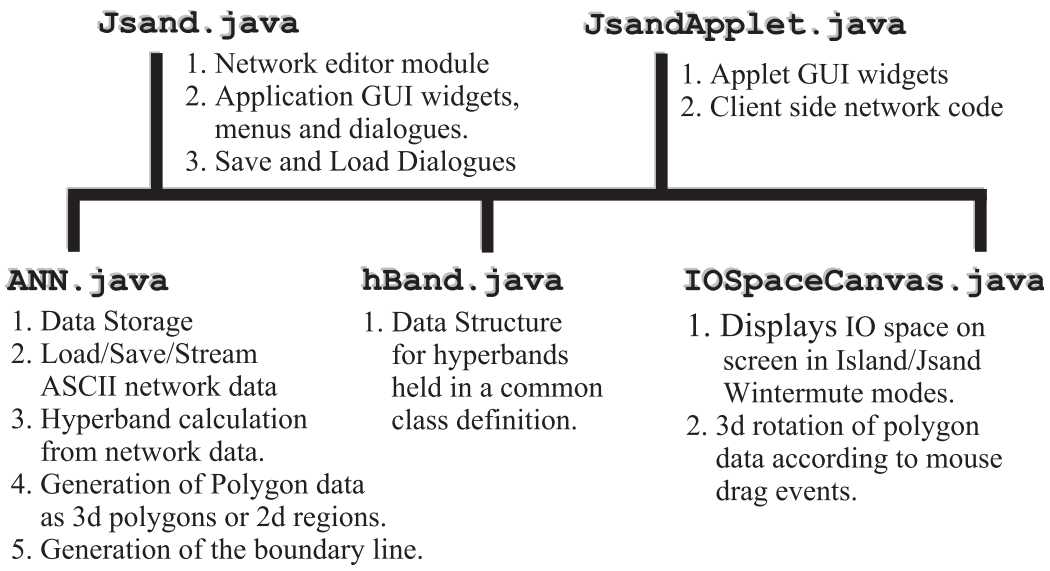


Figure 7.6: The source files comprising JSAND

### 7.1.6 Implementation of JSAND

In most cases each software module in the implementation plan corresponds to a Java class in the final implementation. Since JSAND may be built as both an applet and a Java application separate source files are used in order to allow both the applet and application to share common code such as the display canvas module. The applet and application builds differ mainly in their initialisation and GUI code. Also the application build contains a network editor and GUI functions to load and save a neural network from disc whereas the applet contains client side code required to communicate with a training program running on a remote server.

Figure 7.6 shows the source tree of JSAND used to build both the applet and application along with numbered descriptions of the software modules contained within each source. Compilation of the `Jsand.java` source file and all three sources below it in the tree produce the application, and compilation of the `JsandApplet.java` source along with the three sources below it generates the applet. The Java compiler program will automatically compile sources for modules referenced in the current source as long as they are contained in the current working directory. At run time the Java virtual machine will then dynamically link

each class file to form either a working JSAND applet or application. The three source files common to both applet and application builds provide the following functionality:

1. `ANN.java` contains the data storage module described in the plan spread through three classes, `ANN`, `Neuron` and `Weight`. `ANN` is the central class, which contains constructors and methods able to generate a new random 2-n-1 neural network, or create a new neural network from an ASCII description in the format described previously. The `ANN` class accepts ASCII descriptions for parsing from a Java input stream. This allows neural network data to be streamed from a remote server or loaded from a local file. `ANN` also contains methods to output the current network state to a Java output stream allowing data to be saved to a local file.

   The `Neuron` and `Weight` classes are used by the `ANN` class to represent the data structure of a neural network in an object based manner. The `ANN` class also contains methods to generate 2d regions, 3d polygons, lists of hyperbands - each an `hBand` object - and boundary lines. Once generated, regions, polygons hyperbands and boundary lines are passed to the `IOSpaceCanvas` object for display.

2. `hBand.java` is a simple class definition which is used by both the `IOSpaceCanvas` class and the `ANN` class. When instantiated as an object it represents a single hyperband.

3. `IOSpaceCanvas.java` contains all the code used to plot an IO space display on to a Java AWT canvas. All three display modes - Jsand, Island and Wintermute - are managed by this class, and when instantiated the chosen display mode and attributes may be altered by toggling public flags in the `IOSpaceCanvas` object.

## 7.2 The subgoal chain display

The subgoal chain display applet (see Figure 7.7) allows the changing behaviour of a neural network to be plotted in output space. Since output space is n-dimensional the applet must reduce each n-dimensional point to a two dimensional point as described in the project plan. The two dimensional display contains each n-dimensional subgoal represented by coloured

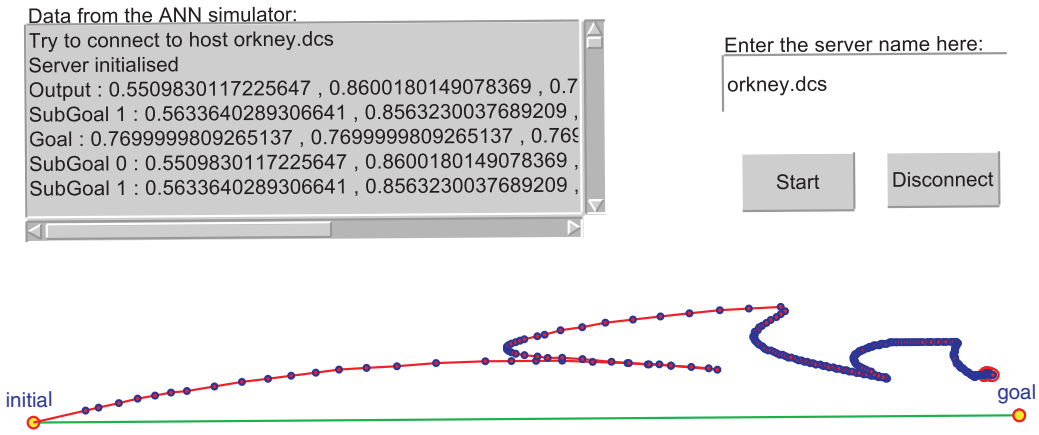


Figure 7.7: An example subgoals applet display

circles, with a set of line segments superimposed. Each line segment represents a change in output state of the network as supervised learning progresses. Together these line segments give a 2d representation of the learning path of the network as it is trained towards each subgoal. The user is then able to visually determine how far from any subgoal the network is at any point in its training.

As with the JSAND applet program the subgoal applet receives training data from a remote server program. The same server program is used, which again simply forwards the ASCII output of the training program to the applet once a network connection has been established. The applet then parses this data stream in order to find new output states for display. The training program being used was supplied by Dr Weir's research group, and makes use of a particular subgoal training algorithm. This training program outputs the following collection of ASCII tags :

- `SG_` Indicates a new subgoal is to follow.
- `O_g_O` = Indicates a new goal state is to follow.
- `w_c_O` = Indicates a new weight state to follow.
- `O_c_O` = Indicates a new output state from the network is to follow.

Each of these textual tags is followed by the appropriate list of comma separated real numbers in ASCII format. Lists of real numbers are delimited by curly braces. The applet parses this data stream and displays the resulting information on the two dimensional display canvas as it is received. Additionally a Java textbox is used to display all elements of the data stream that have been used in creating the display. This textbox contains all output states, subgoals and goals being displayed, which may then be cut and pasted by the user to another application, or examined within the scrollable text box.

One advantage of applet programs is that vector graphics being displayed such as the path of the network may be output as postscript vector graphics. The appletviewer program supplied as an integral part of the Java Development Kit (JDK) allows postscript files to be printed out from the applet window.

### 7.2.1 Implementation of the subgoal chain display applet

The subgoal chain display applet is built from a single Java source file `SubgoalApplet.java` which containing the following class definitions :

- `SubgoalApplet` This class contains all of the applet GUI and widgets. User controls are provided to allow connection to the remote server program. All Java events are handled by this class, and an initial connection to the server is established by the class, before a `SubGoalReceiver` thread is instantiated to handle the connection.

- `SubGoalReceiver` This module contains all code necessary to parse incoming ASCII data from the training program. It then passes newly parsed data to a `SubGoalChain` object and requests that the display is re-painted so that the user may see new data points appear in real time. When instantiated the `SubGoalReceiver` class runs as a separate thread to ensure that the subgoal display can be updated and the GUI still functions while the `SubGoalReceiver` thread awaits new data.

- `SubgoalCanvas` This module provides all graphical display functions, essentially the required Java `paint()` method which redraws the display. The data used to produce

the display is held within a `SubgoalChain` object. To speed up redraw operations the `SubgoalCanvas` object caches lists of `xyPoint` objects that have been previously computed from `ndPoint` objects.

- `SubgoalChain` This is the core data storage module in the applet program. It provides methods used to add new subgoals, goals and output states to display. When instantiated this object must be known to both the current `SubGoalReceiver` object and `SubGoalCanvas` object. This allows the receiver to add newly parsed data, and the canvas object to then display this data when a repaint operation is requested. Data points are stored as lists of `ndPoint` objects.
- `xyPoint` Provides a simple container data structure for a pair of integer values.
- `ndPoint` Provides an abstract data type for manipulating n-dimensional points. An `ndPoint` object may be constructed and then reduced to an `xyPoint` object using methods provided.

## 7.3 The server program

The server program is written in C, and is able to listen to a specific network port for connection requests from either Subgoal or JSAND applets. Once a connection has been established and an appropriate handshake string has been received from the applet, the server program attaches the input stream of the socket connection to the UNIX standard output stream. The server then calls the main function of the training program, and all further output from the training program is forwarded directly to the applet for parsing.

To achieve this functionality the training program must have its original `main()` function renamed to `simulator_initialise()`. Additionally since the training program may read configuration information piped from file via the UNIX standard input stream, this input stream must be attached to before the `simulator_initialise()` call is made. The UNIX `fork()` function is used before calling `simulator_initialise()` so that the server may

continue to listen for more applet connections and fork more training programs as necessary. The training program is statically linked to the server program at compile time.

# Chapter 8

# Critical Appraisal

The JSAND IO space display system provides enhanced functionality over previously developed IO space display systems. JSAND integrates the functionality provided by the Island and Wintermute programs and expands on both with a third analogue three dimensional display method which allows the user to gain a very intuitive feel for the shape of an IO space mapping. In comparing JSAND directly with the Wintermute program however it should be noted that all network activation calculations within JSAND are based on a Hyperband activation function, whereas Wintermute allows a Sigmoidal activation function to be used as well. The original Wintermute program was designed for off line use however, whereas JSAND allows a real time view in Wintermute mode to be generated.

The original ISLAND program provided greater functionality in its ability to edit neural networks directly without training. Storer's Island was able to alter the angle of hyperbands displayed in IO space allowing users to construct networks designed to solve specific problems with no further training. However some of this functionality can be inherited by JSAND because of the closeness of the two program's file format. Minimal changes would be necessary to allow the import of neural networks that had been edited/generated by ISLAND allowing them to be displayed in the superior three dimensional JSAND system. ISLAND is limited to providing two dimensional displays of network activation, and only registers the activity or lack of activity that each neuron provides for a display coordinate.

The subgoal display system relies on new theory produced by Dr Weir at St Andrews, and as such has no directly related previous work to be compared against. The display applet relies on its ability to acquire data in real time from a training program, and as the data stream used is constructed from simple ASCII tokens the applet should be easily interfaced to new training programs developed in the future.

# Chapter 9

# Testing Summary

At implementation each Java class was tested independently, as was each method within the classes. The ability of the Jsand program to produce three different types of output display has been central to the validation of the code used to generate each different IO space display type. Since each display mode (Wintermute, Island and Jsand) is able to render the same data in different styles - each style using different sections of source code - the correct operation of one display mode could be visually verified by comparing it to the display produced by another mode. Particularly the boundary line generation code has benefited from this cross checking. When superimposing the boundary line over a Wintermute mode display, in all randomly generated test neural networks the line has been seen to be in the correct place.

The use of random neural networks as test data sets has enabled the testing of JSAND with a wide range of neural network data, as each time the JSAND program has been run during development, a different data set has been used.

The correct operation of the object based neural network data storage module was verified using many debugging assertions and print statements to standard output. When a Java print statement encounters an object reference that object's memory reference is printed in place. This has made it easy to verify that the object links in the neural network data structures are correct.

## 9.1 The parser

In order to test the functionality of the parser module, a variety of test data sets were generated, and the parsed output from these sets verified by eye. Since the parser could be faced by a number of unexpected input conditions thorough testing of this module was important, both at the level of individual functions and as a whole. The same core parser functions have been used in the JSAND and SubgoalsApplet programs. This code re-use should help reduce the number of problems, as testing time was able to be concentrated to on the initial implementation of the parser instead of three separate parser programs. In the case of JSAND all parser code is held centrally in the `ANN.java` module, a source file common to both the applet and application builds.

# Chapter 10

# User Manual

## 10.1 The JSAND Application

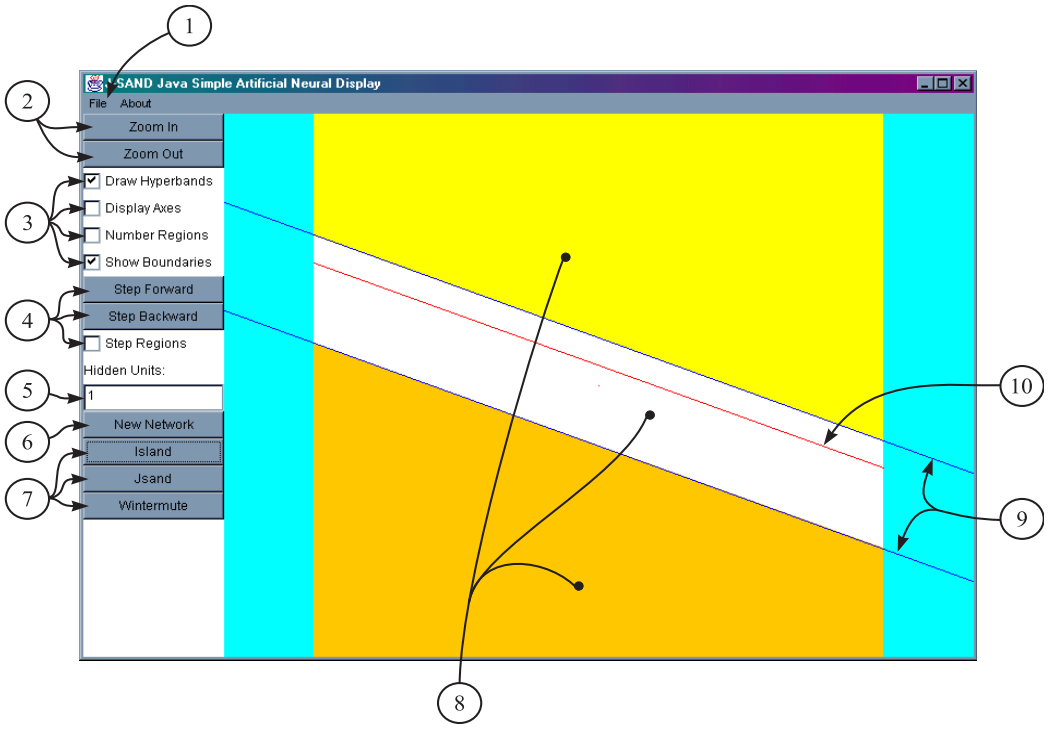


Figure 10.1: The main JSAND application window

Figure 10.1 shows the main JSAND application window. On startup a single hyperband

from a 2-n-1 ANN (where n the number of hidden units is 1) will be displayed. The number of hidden units (n) may be altered by clicking on the hidden units text box (5 in the figure) and entering a new value. To generate a new random network click the `New Network` button (6). One hyperband will be displayed per hidden unit.

When a suitable network is displayed you have a choice of three possible display modes. JSAND initially shows an Island mode display, as in figure 10.1 where one hyperband (9) and three regions (8) are shown. A boundary line (10) is also present. The lower region is coloured orange, indicating that it is on the off side of the hyperband, the upper region yellow as it is on the on side and the region within the hyperband is white. The display mode may be selected using the `Island`, `Jsand` and `Wintermute` buttons (7).

The display may be scaled in any of the three modes using the `Zoom In` and `Zoom Out` buttons (2). Properties of the current display mode can be configured using the four radio buttons (3) below the zoom controls. These buttons allow the optional display of the boundary line, numbering of the regions, hyperbands and axes. In certain display modes some of these options are unavailable, for example region numbering cannot be displayed in Jsand mode. The file menu is selected from the menu bar (1). Figure 10.2 illustrates this menu and the four options provided. The user can load (1) a new network using a platform specific dialogue box, save (3) a network, edit (2) a network and quit (4) the JSAND program. The edit network function produces a dialogue box which allows specific neuron weight values to be changed.

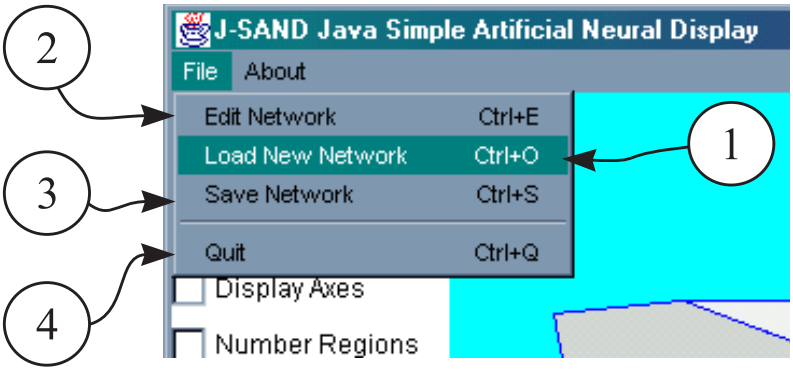


Figure 10.2: The JSAND application file menu

### 10.1.1 The Wintermute display mode

Figure 10.3 shows JSAND in Wintermute display mode, with the boundary line option (4) turned on. The boundary line (1) can be clearly seen dividing the lighter left (3) hand side of IO space from the darker right (2) hand side.

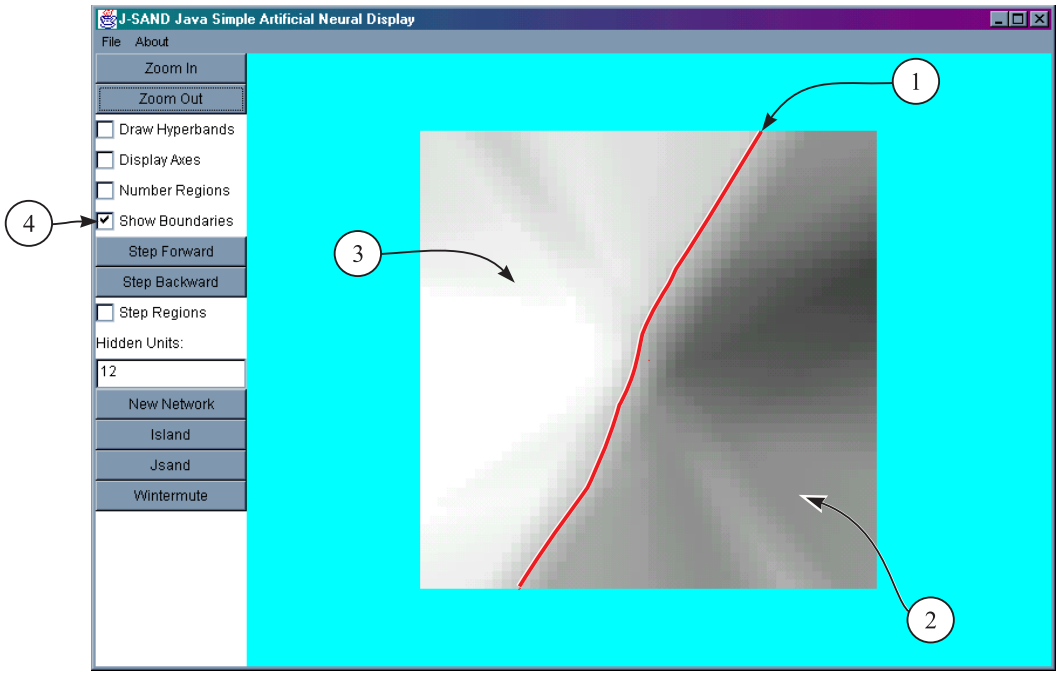


Figure 10.3: The Wintermute display mode

### 10.1.2 The Island display mode

Figure 10.4 shows an Island mode display of two hyperbands (marked 1 and 2) and a boundary line (3).

### 10.1.3 The Jsand display mode

Figure 10.5 shows the same two hyperbands from figure 10.4 in Jsand mode (again marked 1 and 2). This Jsand mode display has been rotated by the user dragging the mouse across the IO space display. As the mouse moves, the display should rotate. The `Step Forward`,

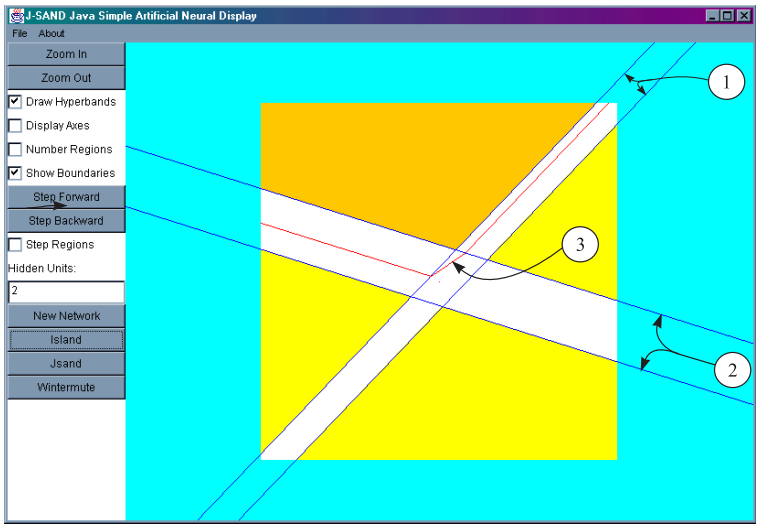


Figure 10.4: An Island mode display of two hyperbands and a boundary line

`Step Back` and `Step Regions` buttons may be used in any of the three display modes to show only one selected region. However in Jsand mode, all regions are still drawn in wire frame with the selected region appearing solid. Screen updates when dragging a Jsand mode display are faster when the `Step Regions` option is selected to enable wire frame plotting.

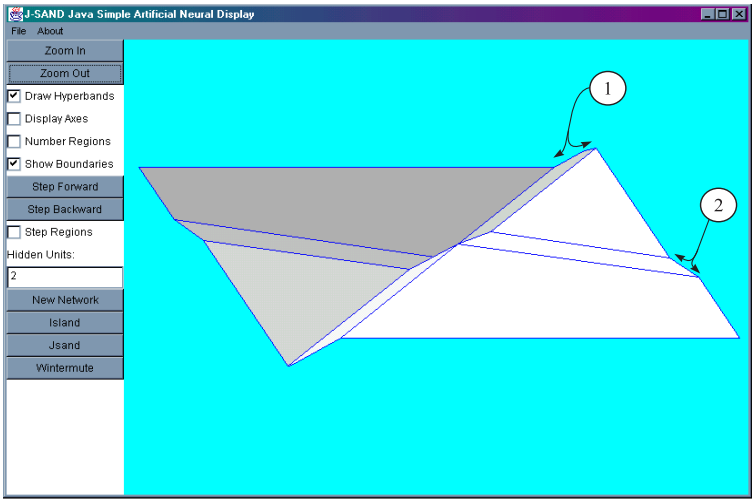


Figure 10.5: An jsand mode display of two hyperbands

## 10.2 The JSAND Applet

The JSAND applet is operated in the same way as the JSAND application. The applet has no file menu, instead having a `Connect to Server` button in the lower part of the window, along with a writeable text box where the user can enter a server host name. Once the name has been entered, and `Connect to Server` pressed the applet will begin streaming training data from the specified server. If no server was available an error message will be printed in the Java console, and no streaming will take place.

## 10.3 The Subgoals Applet

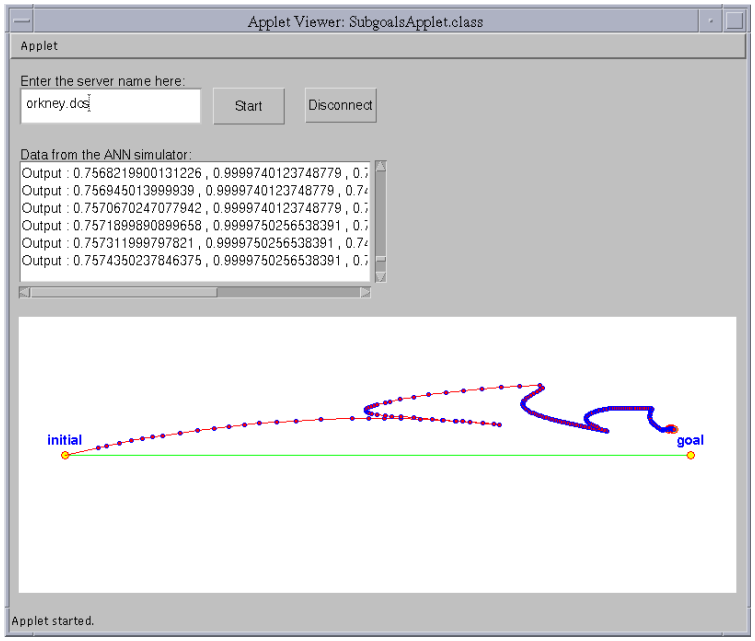


Figure 10.6: The subgoal display applet in a Solaris desktop

Once loaded in a web page, the subgoals applet (see Figure 10.6) may be connected to a server by entering the server name in the server text box, and then clicking the `Connect` button. As with the JSAND applet, data is streamed from the remote server after this action has been taken.

The output path is displayed in the lower part of the applet window. On connection to the server, an initial network state and a goal network state are plotted and marked on screen. From there new subgoal points and output state line segments are added as they are received by the applet. The output and goal states received are also printed in a Java text box on the top left of the applet window. States in the text box may be cut and pasted into other application programs.

## 10.4 The Server Program

The server program must be built along with the training program, as described in the maintenence document. Assuming that the server is built and ready to run, it may be simply started at the Solaris command line by name, for example `SimServ`. If the training program requires data to be piped to it, the command should also include the file and the pipe command, for example `SimServ < NetStates`.

Once run the server will await network connections, and fork a new training program for each connection received from a subgoals applet or a JSAND applet.

# Chapter 11

# Maintenance Document

The Java parts of the software deliverables have been developed using the Java object oriented philosophy, which has allowed most software modules described in the plan to be implemented as separate classes. In some cases, good object oriented programming practice has led to two modules being contained within the same class. For example, the neural network data storage module also contains methods to load and save a network - functions which were indicated as separate software modules in the plan.

## 11.1 Compilation of the source files

The `javac` compiler may be used to compile both applets and the jsand program. Gcc is used to compile the server program, with linking options dependent on the training program being linked to. Example compilation commands for each program are as follows :

- The Jsand application may be built by issuing a `javac Jsand.java` command.
- Similarly the applet is built using `javac JsandApplet.java`.
- The Subgoals applet may be built with `javac SubgoalsApplet.java`

- The Server program is built using gcc with the following command `gcc SimServ.c Trainer.c -lm -lnsl -lsocket` where the `Trainer.c` program is the training program with `main()` renamed `simulator_initialise()`.

For the above commands to work it is assumed that all source files are contained in one working directory.

## 11.2 Alteration of the parsers for different training programs

The parsers used to read ASCII input from a training program may be altered simply for use with different training programs. All tokens used by the parser are held as constants at the top of the class file. Should a training program output have a different output syntax to that expected, minor changes will be needed to the subgoal or the jsand parser code.

In the case of the jsand parser, the structure of neural network data internally is very flexible. Since a neural network module is a Java object, which can have elements (weights and neurons) added to dynamically, the parser is able to build a complete network object before discarding the old one and updating the display. The `ANN` class contains methods to add `weight` and `neuron` objects after the network has been constructed.

## 11.3 Testing strategies used

During construction of the Jsand program the *step through regions* function proved invaluable in graphically debugging the data structures used. Being able to display only one region at a time has allowed me to see clearly when incorrect region data was generated.

The telnet program under solaris was useful in testing the functionality of the server program without using the client applets. Telnet allows connections to be made to the server port, and textual commands to be issued. In this way the correct behaviours of the server on the network could be tested for.

# Chapter 12

# Software Listings

## 12.1 The Jsand Application `Jsand.java`

```
/*****************************************\
 * JSand Hyperband display system (application build)
 *     By Duncan McPherson
 *
 *                5th of March 2000
\*****************************************/

import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.awt.image.*;
import java.lang.Math.*;

public class Jsand extends Frame implements ActionListener, WindowListener,
 MouseListener, MouseMotionListener    {

    // The display canvas and all widgets needed to control it :
    public IOSpaceCanvas display;
    private MenuItem edit, load, save, quit, about;
    private Button zoomin, zoomout, stf, stb, redraw, wintermute, island, threed ;
    private Panel panel1;
    private Checkbox drawbands, numregions, axes, stepon, bounds ;
    TextField numhidden ;
    Label numlabel ;
    EditDialog editor ;
```

```
 ANNLoader loader = null ;

 // Main entry point to the java program – sets up main display
 // window, and positions it on the screen :
 public static void main(String[] args) {
int gap = 200;
int start_x = 100;
int start_y = 50;
Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
Jsand window = new Jsand() ;
window.show();
window.setSize (screen.width – gap, screen.height – gap);
window.setLocation(start_x, start_y);
 }

 // The constructor for the main window :
 public Jsand() {

// Initial size and position of the display
this.setSize ( 800, 600 );
setTitle("J-SAND Java Simple Artificial Neural Display");
addWindowListener(this);
// First add a menubar to the main window :
MenuBar menubar = new MenuBar();
Menu file, tools, about_menu;
file = new Menu ("File", true);
edit = new MenuItem ("Edit Network", new MenuShortcut(KeyEvent.VK_E) );
file.add (edit);
load = new MenuItem ("Load New Network", new MenuShortcut(KeyEvent.VK_O));
file.add (load);
save = new MenuItem ("Save Network", new MenuShortcut(KeyEvent.VK_S));
file.add (save);
file.addSeparator();
quit = new MenuItem ("Quit", new MenuShortcut(KeyEvent.VK_Q));
file.add (quit);
edit.addActionListener(this);
load.addActionListener(this);
save.addActionListener(this);
quit.addActionListener(this);
about_menu = new Menu ("About", true);
about = new MenuItem ("About", new MenuShortcut(KeyEvent.VK_A));
about_menu.add (about);
about.addActionListener(this);
menubar.add (file);
```

```
menubar.add (about_menu);
setMenuBar (menubar);

// Add the control panel and all its buttons to the main window :
Panel panel1 = new Panel();
this.add(panel1, "West");
panel1.setLayout( new GridLayout( 20, 1 ) ) ;
zoomin = new Button("Zoom In");
panel1.add( zoomin );
zoomin.addActionListener(this) ;
      zoomout = new Button("Zoom Out");
zoomout.addActionListener(this) ;
panel1.add( zoomout );
drawbands = new Checkbox("Draw Hyperbands") ;
panel1.add( drawbands ) ;
axes = new Checkbox("Display Axes") ;
panel1.add( axes ) ;
numregions = new Checkbox("Number Regions") ;
panel1.add( numregions ) ;
bounds = new Checkbox("Show Boundaries") ;
panel1.add( bounds ) ;

// Debugging buttons here :
      stf = new Button("Step Forward");
panel1.add( stf ) ;
stf.addActionListener(this) ;
      stb = new Button("Step Backward");
panel1.add( stb ) ;
stb.addActionListener(this) ;
stepon = new Checkbox("Step Regions") ;
panel1.add( stepon ) ;

numlabel = new Label("Hidden Units:");
panel1.add( numlabel ) ;
numhidden = new TextField( 10 );
      numhidden.setEditable(true);
panel1.add( numhidden );
redraw = new Button("New Network");
panel1.add( redraw ) ;
redraw.addActionListener(this) ;
island = new Button("Island");
panel1.add( island ) ;
island.addActionListener(this) ;
threed = new Button("Jsand") ;
```

```
panel1.add( threed ) ;
threed.addActionListener(this) ;
wintermute = new Button("Wintermute");
panel1.add( wintermute ) ;
wintermute.addActionListener(this) ;

// Add the canvas to the main window, and initialise the network :
numhidden.setText("1") ;
display = new IOSpaceCanvas() ; // And an IO space display canvas
display.network = new ANN( 1 ) ; // Create an ANN with x hidden units
display.bands = display.network.calculateHyperbands() ;
display.regions = display.network.calculateRegions() ;
display.boundaries = display.network.calculateBoundaries( display.regions ) ;
display.draw_bands = true ;
drawbands.setState( true ) ;
add( display ) ;
display.addMouseListener(this) ;
display.addMouseMotionListener(this) ;          }

 public void actionPerformed(ActionEvent event) {
Object source = event.getSource();

if (source == quit) {
     System.exit(0);
} else if ( source == about ) {
     AboutDialog a_dialog = new AboutDialog(this, "About JSAND");
} else if ( source == edit ) {
     // Create an ANN with the number of specified hidden units
     editor = new EditDialog( display, this );
} else if ( source == load ) {
     // Create an ANN with the number of specified hidden units
     FileDialog dialog = new FileDialog(this, "Load Neural Network",
      FileDialog.LOAD );
     dialog.show();
     String file = "" + dialog.getFile();
     if (!file.equals ("") && !file.equals("null")) {
    if ( loader != null )
         loader.loadThread.stop() ;
    loader = new ANNLoader( display, dialog.getDirectory() + file, numhidden ) ;
     }
} else if ( source == save ) {
     // Create an ANN with the number of specified hidden units
     FileDialog dialog = new FileDialog(this, "Save Neural Network",
      FileDialog.SAVE );
```

```
        dialog.show();
        String file = "" + dialog.getFile();
        // Some file has been selected and not 'cancel'
        if (!file.equals ("") && !file.equals("null"))
       display.network.saveNetwork(      dialog.getDirectory() + file ) ;
    } else if ( source == zoomout && display.zoom > 0.25) {
        display.zoom = display.zoom – 0.25 ;
        display.repaint() ;
    } else if ( source == zoomin && display.zoom < 25 ) {
        display.zoom = display.zoom + 0.25 ;
        display.repaint() ;
    } else if ( source == stf ) {
        if ( ! display.step ) {
       display.stnum = 0 ;
       display.step = true ;
       stepon.setState( true ) ;
        } else {
       display.stnum++ ;
       if ( display.stnum >= display.regions.length )
            display.stnum = 0 ;
        }
        display.repaint() ;
    } else if ( source == stb ) {
        if ( ! display.step ) {
       display.stnum = 0 ;
       display.step = true ;
       stepon.setState( true ) ;
        } else {
       display.stnum–– ;
       if ( display.stnum < 0 )
            display.stnum = display.regions.length–1 ;
        }
        display.repaint() ;
    } else if ( source == redraw ) {
        if ( loader != null )
       loader.loadThread.stop() ;
        display.network = new ANN( Integer.parseInt( numhidden.getText(), 10 ) ) ;
        display.bands = display.network.calculateHyperbands() ;
        display.regions = display.network.calculateRegions() ;
        display.translated = null ;
        display.boundaries = display.network.calculateBoundaries( display.regions ) ;
        display.planes = display.network.calculatePlanes() ;
        display.region_activation = false ;
        display.repaint() ;
```

```
} else if ( source == threed ) {
    display.planes = display.network.calculatePlanes() ;
    display.threed = true ;
    display.xrot = 0 ;
    display.yrot = 90 ;
    display.translated = null ;
    display.repaint() ;
} else if ( source == island ) {
    display.regions = display.network.calculateRegions() ;
    display.region_activation = false ;
    display.threed = false ;
    display.repaint() ;
} else if ( source == wintermute ) {
    display.regions = display.network.calculateWintermute() ;
    display.region_activation = true ;
    display.threed = false ;
    display.repaint() ;
}
 }

 public boolean action( Event evt, Object arg ) {

if ( evt.target.equals(drawbands) ) {
    display.draw_bands = drawbands.getState() ;
    display.repaint() ;
    return true ;
} else if ( evt.target.equals(numregions) ) {
    display.number_regions = numregions.getState() ;
    display.repaint() ;
    return true ;
} else if ( evt.target.equals(axes) ) {
    display.draw_axes = axes.getState() ;
    display.repaint() ;
    return true ;
} else if ( evt.target.equals(stepon) ) {
    display.step = stepon.getState() ;
    display.repaint() ;
    return true ;
} else if ( evt.target.equals(bounds) ) {
    display.boundary_lines = bounds.getState() ;
    display.repaint() ;
    return true ;
}  return super.action( evt, arg ) ;
 }
```

```
    public void windowClosed(WindowEvent event) {
   System.exit(0);
    }

    public void windowClosing(WindowEvent event) {
   System.exit(0);
    }

    public void mouseDragged( MouseEvent e ) {
   if ( display != null )
       display.drag( e.getX(), e.getY() ) ;
    }

    public void mouseReleased( MouseEvent e ) {
   if ( display != null )
       display.drag_finished( e.getX(), e.getY() )          ;
    }

    // Stubs required by java for event handlers
    public void windowDeiconified(WindowEvent event) {}
    public void windowIconified(WindowEvent event) {}
    public void windowActivated(WindowEvent event) {}
    public void windowDeactivated(WindowEvent event) {}
    public void windowOpened(WindowEvent event) {}
    public void mouseClicked(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}

}

class AboutDialog extends Dialog implements ActionListener, WindowListener {
    Button ok_button;     // The button to dispose of the dialog box.
    Panel panel;          // The panel containing the text.

    public AboutDialog(Frame parent_frame, String title) {
   super(parent_frame, title, false) ;
   setSize( 320 , 225 );
   addWindowListener (this);
   panel = new Panel();
   panel.add (new Label ("          Artificial Neural Network      ",
    Label.CENTER));
```

```
    panel.add (new Label (“          Input Output Space Display     ”,
     Label.CENTER));
    panel.add (new Label (“            By Duncan McPherson          ”,
     Label.CENTER));
    panel.add (new Label (“    Senior Honours Project April 2000   ”,
     Label.CENTER));
    ok_button = new Button (“Close About Box”);
    ok_button.addActionListener(this);
    add (“South”, ok_button);
    add (“Center”, panel);
    setLocation ( 200 , 150 );
    show();
     }

     public void actionPerformed(ActionEvent event) {
    if (event.getSource() == ok_button) {
        dispose();
    }
     }

     public void windowClosed(WindowEvent event) {
    dispose();
     }

     public void windowClosing(WindowEvent event) {
    dispose();
     }

     public void windowDeiconified(WindowEvent event) {}
     public void windowIconified(WindowEvent event) {}
     public void windowActivated(WindowEvent event) {}
     public void windowDeactivated(WindowEvent event) {}
     public void windowOpened(WindowEvent event) {}
}

class EditDialog extends Dialog implements ActionListener, WindowListener {
     Button ok, close;       // The button to dispose of the dialog box.
     Panel panel;         // The panel containing the text.
     IOSpaceCanvas display ;
     TextField biasval ;
     Label biaslabel ;

     public EditDialog( IOSpaceCanvas being_displayed, Frame parent_frame ) {
    super(parent_frame, “Edit Network Properties”, false);
```

```
display = being_displayed ;
setSize( 400, 225 );
addWindowListener (this);
panel = new Panel();
panel.setLayout( new GridLayout( 5, 2 ) ) ;
biaslabel = new Label("Bias value:");
panel.add( biaslabel ) ;
biasval = new TextField( 10 );
      biasval.setEditable(true);
panel.add( biasval );
ok= new Button ("Ok");
ok.addActionListener(this);
close= new Button ("Close");
close.addActionListener(this);
panel.add ( ok );
panel.add ( close ) ;
add ( panel ) ;
biasval.setText(""+being_displayed.network.bias ) ;
setLocation ( 200, 150 );
show();
 }

 public void actionPerformed(ActionEvent event) {
if (event.getSource() == close ) {
    dispose();
} else if ( event.getSource() == ok ) {
    display.network.bias = (new Double( biasval.getText() )).doubleValue() ;
    dispose() ;
}
 }

 public void windowClosed(WindowEvent event) {
dispose();
 }

 public void windowClosing(WindowEvent event) {
dispose();
 }

 public void windowDeiconified(WindowEvent event) {}
 public void windowIconified(WindowEvent event) {}
 public void windowActivated(WindowEvent event) {}
 public void windowDeactivated(WindowEvent event) {}
 public void windowOpened(WindowEvent event) {}
```

```
}

/*********************************************\
 * Class : ANNLoader
 *
 * Purpose : Manages all I/O between Applet and ANN Simulator
 *           and an input file that may contain one or more
 *           network descriptions. If more than one they are
 *           loaded to form an animated sequence.
\*********************************************/
class ANNLoader implements Runnable {
    public Thread loadThread; // This thread, which must be active to recieve
     bytes
    IOSpaceCanvas display ;
    public String filename ;
    public TextField units ;

    public ANNLoader( IOSpaceCanvas thedisplay, String input, TextField numhidden
     ) {
   // Copy the instance of the reciever object, and set up buffer
   // variables and flags associated with decoding incoming frames ...
   display = thedisplay ;
   filename = input ;
   units = numhidden ;
   loadThread = new Thread(this);
   loadThread.start();
    }

    // Entry point for the thread on start of execution :
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}

   // Try to load a description of a neural network from the stream
   Vector description = null ;
   try {
       FileInputStream fin = new FileInputStream( filename ) ;
       DataInputStream is = new DataInputStream( fin ) ;
       description = display.network.loadNetwork( is ) ;
       while ( description != null ) {
      display.network.generateNetwork( description ) ;
      if ( display.network.loadedCorrectly ) {
          display.bands = display.network.calculateHyperbands() ;
```

```
            display.regions = display.network.calculateRegions() ;
            display.boundaries = display.network.calculateBoundaries( display.regions )
             ;
            if ( display.region_activation )
           display.regions = display.network.calculateWintermute() ;
            display.translated = null ;
            display.planes = display.network.calculatePlanes() ;
            units.setText( new String( ""+display.network.Hidden.length ) ) ;
            display.repaint() ;
            try {
              Thread.sleep(500);
            } catch (InterruptedException e) {}
            description = display.network.loadNetwork( is ) ;
        } else {
            description = null ;
            display.network = null ;
        }
         }

         fin.close() ;
    } catch( Exception e ) { System.out.println("File error during load"+e) ; }
     }
}
```

## 12.2 The JSAND applet JsandApplet.java

```
/*****************************************\
 *    JSand Hyperband display system (applet build)
 *     By Duncan McPherson
 *
 *                5th of March 2000
\*****************************************/

import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.awt.image.*;
import java.applet.*;
import java.io.*;
import java.net.*;
```

```
public class JsandApplet extends Applet implements ActionListener, MouseListener,
 MouseMotionListener {

    // The display canvas and all widgets needed to control it :
    public IOSpaceCanvas display;
    private MenuItem open, close, quit, about;
    private Button zoomin, zoomout, stf, stb, redraw, wintermute, island, threed,
     connect, disconnect ;
    private Panel panel1;
    private Checkbox drawbands, numregions, axes, stepon, bounds ;
    TextField numhidden, server ;
    Label numlabel ;
    boolean laidOut = false ;

    // Communications with applet server variables :
    DataInputStream is = null ;
    Socket s = null;
    PrintStream os = null ;
    private ANNReceiver receiver ;

    // Main entry point to the applet program – sets up main display
    // area within the applet viewer. No menu bar is created for the applet :
    public void init() {

   display = new IOSpaceCanvas() ; // And an IO space display canvas
   display.network = new ANN( 1 ) ; // Create an ANN with x hidden units
   display.bands = display.network.calculateHyperbands() ;
   display.regions = display.network.calculateRegions() ;
   display.boundaries = display.network.calculateBoundaries( display.regions ) ;
   display.draw_bands = true ;
   this.add( display, "West" ) ;
   Insets insets = insets();
   display.reshape( insets.left, insets.top, 500, 500 ) ;
   display.addMouseListener(this) ;
   display.addMouseMotionListener(this) ;

   // Add the control panel and all its buttons to the main window :
   Panel panel1 = new Panel();
   this.add(panel1, "East");
   panel1.setLayout( new GridLayout( 17, 1 ) ) ;
   zoomin = new Button("Zoom In");
   panel1.add( zoomin );
   zoomin.addActionListener(this) ;
        zoomout = new Button("Zoom Out");
```

```
zoomout.addActionListener(this) ;
panel1.add( zoomout );
drawbands = new Checkbox("Draw Hyperbands") ;
panel1.add( drawbands ) ;
axes = new Checkbox("Display Axes") ;
panel1.add( axes ) ;
numregions = new Checkbox("Number Regions") ;
panel1.add( numregions ) ;
bounds = new Checkbox("Show Boundaries") ;
panel1.add( bounds ) ;

// Debugging buttons here :
      stf = new Button("Step Forward");
panel1.add( stf ) ;
stf.addActionListener(this) ;
      stb = new Button("Step Backward");
panel1.add( stb ) ;
stb.addActionListener(this) ;
stepon = new Checkbox("Step Regions") ;
panel1.add( stepon ) ;

// Display type buttons here :
numlabel = new Label("Hidden Units:");
panel1.add( numlabel ) ;
numhidden = new TextField( 10 );
      numhidden.setEditable(true);
panel1.add( numhidden );
redraw = new Button("New Network");
panel1.add( redraw ) ;
redraw.addActionListener(this) ;
island = new Button("Island");
panel1.add( island ) ;
island.addActionListener(this) ;
threed = new Button("Jsand") ;
panel1.add( threed ) ;
threed.addActionListener(this) ;
wintermute = new Button("Wintermute");
panel1.add( wintermute ) ;
wintermute.addActionListener(this) ;

Panel panel2 = new Panel();
this.add(panel2, "South");
connect = new Button("Connect to Server");
panel2.add( connect ) ;
```

```
connect.addActionListener(this) ;
server = new TextField( 40 );
      server.setEditable(true);
panel2.add( server );
disconnect = new Button("Disconnect");
panel2.add( disconnect ) ;
disconnect.addActionListener(this) ;

// Add the canvas to the main window, and initialise the network :
numhidden.setText("1") ;
drawbands.setState( true ) ;

      validate();
repaint() ;
 }

 public void actionPerformed(ActionEvent event) {
Object source = event.getSource();

if (source == quit) {
     System.exit(0);
} else if ( source == about ) {
     //AboutDialog a_dialog = new AboutDialog(this, "About HyperBands");
} else if ( source == open ) {
     display.network = new ANN( Integer.parseInt( numhidden.getText(), 10 ) ) ;
     display.bands = display.network.calculateHyperbands() ;
     display.regions = display.network.calculateRegions() ;
     display.boundaries = display.network.calculateBoundaries( display.regions ) ;
     display.region_activation = false ;
     display.repaint() ;
} else if ( source == zoomout && display.zoom > 0.25) {
     display.zoom = display.zoom - 0.25 ;
     display.repaint() ;
} else if ( source == zoomin && display.zoom < 25 ) {
     display.zoom = display.zoom + 0.25 ;
     display.repaint() ;
} else if ( source == stf ) {
     if ( ! display.step ) {
    display.stnum = 0 ;
    display.step = true ;
    stepon.setState( true ) ;
     } else {
    display.stnum++ ;
    if ( display.stnum >= display.regions.length )
```

```
            display.stnum = 0 ;
        }
        display.repaint() ;
    } else if ( source == stb ) {
        if ( ! display.step ) {
       display.stnum = 0 ;
       display.step = true ;
       stepon.setState( true ) ;
        } else {
       display.stnum-- ;
       if ( display.stnum < 0 )
            display.stnum = display.regions.length-1 ;
        }
        display.repaint() ;
    } else if ( source == redraw ) {
         // Create an ANN with the number of specified hidden units
        display.network = new ANN( Integer.parseInt( numhidden.getText(), 10 ) ) ;
        display.bands = display.network.calculateHyperbands() ;
        display.regions = display.network.calculateRegions() ;
        display.boundaries = display.network.calculateBoundaries( display.regions ) ;
        display.region_activation = false ;
        display.planes = display.network.calculatePlanes() ;
        display.translated = null ;
        display.repaint() ;
    } else if ( source == threed ) {
        display.planes = display.network.calculatePlanes() ;
        display.threed = true ;
        display.xrot = 0 ;
        display.yrot = 90 ;
        display.translated = null ;
        display.repaint() ;
    } else if ( source == wintermute ) {
        display.regions = display.network.calculateWintermute() ;
        display.region_activation = true ;
        display.threed = false ;
        display.repaint() ;
    } else if ( source == island ) {
        display.regions = display.network.calculateRegions() ;
        display.threed = false ;
        display.region_activation = false ;
        display.repaint() ;
    } else if ( source == disconnect ){
        // Disconnect from the server here :
        disconnect() ;
```

```
}  else if ( source == connect ){

     if ( receiver != null ) {
    receiver.receiveThread.stop() ;
    if ( s != null ) disconnect() ;
     }

     String host = server.getText();

     // Attempt to connect to the given server name and then
     // involk the ANN simulator :
     try {
    s = new Socket(host, 5555 ) ;
    is = new DataInputStream(s.getInputStream() ) ;
    os = new PrintStream( s.getOutputStream() ) ;
    os.print("ANNSIMSERVER") ;
    System.out.println("Server initialised\n") ;

    // Create a new thread which handles all
    // output from the newly connected server :
    receiver = new ANNReceiver( display, is, numhidden, this ) ;

     } catch(Exception ex) {
    // System.out.println("Error" + ex) ;
    System.out.println("Unable to connect to "+host+"\n") ;
     }
}
 }

 public void disconnect() {
if ( s != null )
     try {
    s.close() ;
     } catch( Exception ex ) {}
s = null ;
 }

 public boolean action( Event evt, Object arg ) {

if ( evt.target.equals(drawbands) ) {
     display.draw_bands = drawbands.getState() ;
     display.repaint() ;
     return true ;
} else if ( evt.target.equals(numregions) ) {
```

```
        display.number_regions = numregions.getState() ;
        display.repaint() ;
        return true ;
    } else if ( evt.target.equals(axes) ) {
        display.draw_axes = axes.getState() ;
        display.repaint() ;
        return true ;
    } else if ( evt.target.equals(stepon) ) {
        display.step = stepon.getState() ;
        display.repaint() ;
        return true ;
    } else if ( evt.target.equals(bounds) ) {
        display.boundary_lines = bounds.getState() ;
        display.repaint() ;
        return true ;
    }
    return super.action( evt, arg ) ;
     }

     public void mouseDragged( MouseEvent e ) {
    if ( display != null )
        display.drag( e.getX(), e.getY() ) ;
     }

     public void mouseReleased( MouseEvent e ) {
    if ( display != null )
        display.drag_finished( e.getX(), e.getY() )          ;
     }
     public void windowClosed(WindowEvent event) {
    System.exit(0);
     }

     // Stubs required by java for event handlers
     public void windowDeiconified(WindowEvent event) {}
     public void windowIconified(WindowEvent event) {}
     public void windowActivated(WindowEvent event) {}
     public void windowDeactivated(WindowEvent event) {}
     public void windowOpened(WindowEvent event) {}
     public void mouseClicked(MouseEvent e) {}
     public void mouseMoved(MouseEvent e) {}
     public void mouseEntered(MouseEvent e) {}
     public void mouseExited(MouseEvent e) {}
     public void mousePressed(MouseEvent e) {}
}
```

```
/*********************************************\
 * Class : ANNReceiver
 *
 * Purpose : Manages all I/O between Applet and ANN Simulator
 *           which is running remotely. Communication via BSD
 *           sockets. This class executes as a standalone
 *           thread in paralell with the rest of the applet
\*********************************************/

class ANNReceiver implements Runnable {
    public Thread receiveThread; // This thread, which must be active to recieve
     bytes
    private DataInputStream inputStream; // Holds the input byte stream from
     reciever
    public TextField units ;
    public IOSpaceCanvas display ;
    public JsandApplet applet ;

    public ANNReceiver( IOSpaceCanvas thedisplay, DataInputStream input,
          TextField numhidden, JsandApplet theapplet ) {
   // Copy the instance of the reciever object, and set up buffer
   // variables and flags associated with decoding incoming frames ...
   display = thedisplay ;
         inputStream = input ;
   units = numhidden ;
   applet = theapplet ;
   receiveThread = new Thread(this);
   receiveThread.start();
    }

    // Entry point for the thread on start of execution :
    public void run() {
         try {
              Thread.sleep(1000);
         } catch (InterruptedException e) {}
   // Try to load a description of a neural network from the stream
   Vector description = null ;
   description = display.network.loadNetwork( inputStream ) ;
   while ( description != null ) {
        display.network.generateNetwork( description ) ;
        if ( display.network.loadedCorrectly ) {
       display.bands = display.network.calculateHyperbands() ;
```

```
        display.regions = display.network.calculateRegions() ;
        display.boundaries = display.network.calculateBoundaries( display.regions ) ;
        if ( display.region_activation )
            display.regions = display.network.calculateWintermute() ;
        display.translated = null ;
        display.planes = display.network.calculatePlanes() ;
        units.setText( new String( ""+display.network.Hidden.length ) ) ;
        display.repaint() ;
        try {
            Thread.sleep(500); // Half a second delay between frames
        } catch (InterruptedException e) {}
        description = display.network.loadNetwork( inputStream ) ;
         } else {
        description = null ;
        display.network = null ;
         }
    }
    applet.disconnect() ;
     }
}
```

## 12.3 The neural network storage module ANN.java

```
/*****************************************\
 *            JSand ANN Data Module
 * By Duncan McPherson
 *
 * The ANN data module allows neural networks to be
 * created, stored, manipulated, loaded and saved and
 * also transformed into data useable for display
\*****************************************/

import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.awt.image.*;
import java.lang.Math.*;

// Class used to represent a 2-n-1 neural network, making use
// of ancilliary classes neuron and weight to create the data
// structure :
```

```
class ANN {

    private Neuron[] Inputs ;
    public Neuron[] Hidden ;
    private Neuron Output ;
    public double bias = 3 ;
    public double kconstant = 2.5 ;

    // Constructor function used to create the ANN
    // Topology – 2 inputs, n hidden and one output
    // hidden neurons
    public ANN( int num_hidden ) {

        // Initialise a random number generator to create weight values :
    java.util.Random generator = new java.util.Random( System.currentTimeMillis() )
    ;

    // Generate two input units (since they are input units their inputs are null)
    Inputs = new Neuron[2] ;
    Inputs[0] = new Neuron( null, generator ) ;
    Inputs[1] = new Neuron( null, generator ) ;

    // Generate 'n' hidden units, and connect each to the two input units
    Hidden = new Neuron[num_hidden] ;
    int counter ;
    for ( counter = 0 ; counter < num_hidden ; counter++ )
        Hidden[counter] = new Neuron( Inputs, generator ) ;

    // Finally generate a single output unit
    Output = new Neuron( Hidden, generator ) ;
    }

    // Tokens used in the file format :
    public final static String NURN = "_Neuron" ;
    public final static String NTYP = "Type" ;
    public final static String FROM = "From Unit:" ;
    public final static String TO   = "To Unit:" ;
    public final static String WGHT = "Weight:" ;
    public final static String DONE = "Done" ;
    public final static int INPUTNEURON = 1 ;
    public final static int OUTPUTNEURON = 2 ;
    public final static int HIDDENNEURON = 3 ;
    public final static int BIASNEURON = 7 ;
```

```
 // The neural network file parsar – Parses a complete neural network
 // data structure and returns only the strings in the file that
 // pertain to that structure (so that a stream can have one or more
 // data structures parsed by multiple calls to parseNetwork).
 public boolean loadedCorrectly = true ;
 int biasNeuron = 0 ; // Three results from the preparsing :
 int max = 0 ;
 int num_hidden = 0 ;
 public Vector loadNetwork( DataInputStream is ) {
Vector netDescription = new Vector() ;
int inputParsed = 0 ; // Ensure the network is indeed
int outputParsed = 0 ; // a valid 2–n–1 neural network
int neuron = –1 ;
num_hidden = 0 ;
biasNeuron = –1 ;
boolean result = false ;
boolean foundDone = false ;
String temp ;

// First ensure that the input stream contains a network, and then
// pass all the relevent lines from the input stream to the generator :
try { temp = is.readLine() ; } catch( Exception e ) { temp = null ; }
while ( temp != null && !foundDone ) {
    temp = temp.trim() ;
    if ( temp.startsWith( NURN ) ) {
   netDescription.addElement( (Object)new String( temp ) ) ;
   temp = temp.substring( NURN.length(), temp.length() ) ;
   neuron = Integer.parseInt(temp.trim(), 10);
   if ( neuron > max ) max = neuron ;
    } else if ( temp.startsWith( NTYP ) ) {
   netDescription.addElement( (Object)new String( temp ) ) ;
   temp = temp.substring( NTYP.length(), temp.length() ) ;
   int ntype = Integer.parseInt(temp.trim(), 10);
   if ( ntype == INPUTNEURON ) inputParsed ++ ;
   if ( ntype == HIDDENNEURON ) num_hidden ++ ;
   if ( ntype == OUTPUTNEURON ) outputParsed ++ ;
   if ( ntype == BIASNEURON ) biasNeuron = neuron ;
    } else if ( temp.startsWith( FROM ) ) {
   netDescription.addElement( (Object)new String( temp ) ) ;
    } else if ( temp.startsWith( TO ) ) {
   netDescription.addElement( (Object)new String( temp ) ) ;
    } else if ( temp.startsWith( DONE ) ) {
   foundDone = true ;
    }
```

```
    try { temp = is.readLine() ; } catch( Exception e ) { temp = null ; }
}

// If what we have parsed forms a complete network, then generate
// all network data structures that are needed :
if ( inputParsed != 2 || outputParsed != 1 || num_hidden <=0 )
    netDescription = null ;

return netDescription ;
 }

 // Constructs a 2–n–1 network data structure from a textual description
 // Makes use of the class wide variables max, biasNeuron and hiddenParsed
 public boolean generateNetwork( Vector description ) {
Inputs = new Neuron[2] ;
Hidden = new Neuron[num_hidden] ;
Output = new Neuron() ;
//System.out.println("Maxneuron = "+max) ;

Neuron[] newNeurons = new Neuron[max+1] ;
int input = –1 ;
int hidden = –1 ;
int neuron = –1 ;
// Generate all the required neuron objects, then connect them
// with weight objects and values parsed from the description file
int counter ;
for ( counter = 0 ; counter < newNeurons.length ; counter ++ )
    newNeurons[counter] = new Neuron() ;

for ( counter = 0 ; counter < description.size() ; counter++ ) {
    String temp = (String)description.elementAt(counter) ;
    if ( temp != null ) {
   if ( temp.startsWith( NURN ) ) {
        temp = temp.substring( NURN.length(), temp.length() ) ;
        neuron = Integer.parseInt(temp.trim(), 10);
   } else if ( temp.startsWith( NTYP ) ) {
        temp = temp.substring( NTYP.length(), temp.length() ) ;
        int ntype = Integer.parseInt(temp.trim(), 10);
        if ( ntype == INPUTNEURON )
       Inputs[++input] = newNeurons[neuron] ;
        else if ( ntype == OUTPUTNEURON )
       Output = newNeurons[neuron] ;
        else if ( ntype == HIDDENNEURON )
       Hidden[++hidden] = newNeurons[neuron] ;
```

```
    } else if ( temp.startsWith( FROM ) ) {
        temp = temp.substring(FROM.length(), temp.length() ).trim() ;
        int btwn = Integer.parseInt(temp.substring( 0, temp.indexOf(" ")).trim(),
         10);
        temp = temp.substring(temp.indexOf(WGHT)+WGHT.length(),
         temp.length()).trim() ;
        double weight = findDouble( temp ) ;
        if ( neuron != biasNeuron && btwn != biasNeuron )
       newNeurons[btwn].addWeight( newNeurons[neuron], weight ) ;
        if ( btwn == biasNeuron )
       newNeurons[neuron].biasWeight = weight ;
    }
     }
}
return true ;
 }

 // Yes I know this function is strange - workaround
 // for problems with java maths libraries :
 public double findDouble( String in ) {
if ( in.startsWith( "0." ) )
    in = "+"+in ;
Double result = new Double( in ) ;
if ( result.isNaN() ) result = new Double( in ) ;
return result.doubleValue() ;
 }

 // Saves a description of a 2-n-1 neural network to file :
 public void saveNetwork( String filename ) {
System.out.println("save network :"+filename) ;
try {
    FileOutputStream fout = new FileOutputStream( filename ) ;
    PrintStream pout = new PrintStream( fout ) ;
    // Input layer neurons
    int ncount, wcount ;
    for ( ncount = 0 ; ncount < Inputs.length ; ncount++ ) {
   Neuron n = Inputs[ncount] ;
   pout.println( NURN+" "+numberNeuron( n ) );
   pout.println( NTYP+" "+INPUTNEURON );
   for ( wcount = 0 ; wcount < n.outputWeights.size() ; wcount++ ) {
        Weight w = (Weight)n.outputWeights.elementAt(wcount) ;
        pout.println( TO+" "+numberNeuron( w.to )+" "+WGHT+" "+w.weight )
         ;
   }
```

```
        }
        // Hidden layer neurons
        for ( ncount = 0 ; ncount < Hidden.length ; ncount++ ) {
      Neuron n = Hidden[ncount] ;
      pout.println( NURN+" "+numberNeuron( n ) );
      pout.println( NTYP+" "+HIDDENNEURON );
      for ( wcount = 0 ; wcount < n.inputWeights.size() ; wcount++ ) {
          Weight w = (Weight)n.inputWeights.elementAt(wcount) ;
          pout.println( FROM+" "+numberNeuron( w.from )+" "+WGHT+" "
           +w.weight ) ;
      }
      pout.println( FROM+" "+(Inputs.length+Hidden.length+1)+" "+WGHT+" "
       +n.biasWeight ) ;
        }
        // Output neuron
        pout.println( NURN+" "+numberNeuron( Output ) );
        pout.println( NTYP+" "+OUTPUTNEURON );
        for ( wcount = 0 ; wcount < Output.inputWeights.size() ; wcount++ ) {
      Weight w = (Weight)Output.inputWeights.elementAt(wcount) ;
      pout.println( FROM+" "+numberNeuron( w.from )+" "+WGHT+" "+w.weight
       ) ;
        }
        pout.println( FROM+" "+(Inputs.length+Hidden.length+1)+" "+WGHT+" \
"+Output.biasWeight ) ;
        // Bias neuron
        pout.println( NURN+" "+(Inputs.length+Hidden.length+1) );
        pout.println( NTYP+" "+BIASNEURON );
        for ( ncount = 0; ncount < Hidden.length ; ncount++ )
      pout.println( TO+" "+(ncount+Inputs.length)+" "+WGHT+" "
       +Hidden[ncount].biasWeight ) ;
        pout.println( TO+" "+(Inputs.length+Hidden.length)+" "+WGHT+" "
         +Output.biasWeight ) ;

        fout.close() ;
    } catch( Exception e ) { System.out.println("File error on save : "+e) ; }
     }

     // Give a neuron object in the 2–n–1 network a unique integer identifier
     private int numberNeuron( Neuron n ) {
    int counter, result = –1 ;
    for ( counter = 0 ; counter < Inputs.length ; counter++ )
        if ( n.equals( Inputs[counter] ) ) result = counter ;
    if ( result < 0 )
        for ( counter = 0 ; counter < Hidden.length ; counter++ )
```

```
    if ( n.equals( Hidden[counter] ) ) result = counter+Inputs.length ;
if ( result < 0 && n.equals( Output ) )
     result = Inputs.length+Hidden.length ;
return result ;
 }

 public double[] calculateBoundaries( Object[] regions ) {
Vector lines = new Vector() ;
int rgn, h, vertex ;
double x1,y1,x2,y2,ix,iy,grad, x_weight, y_weight, excitation = 0.0 ;
boolean p1above, p2above ;

// For each region, calculate a boundary line :
for ( rgn = 0 ; rgn < regions.length ; rgn++ ) {
     int intersect = 0 ;
     double[] i = new double[4] ;
     double xterm = 0.0 ;
     double yterm = 0.0 ;
     double cterm = Output.biasWeight   ;
     double[] region = (double[])regions[rgn] ;
     double[] xy = regionCentre(region) ;

     for (h=0; h<Hidden.length; h++) {
    x_weight = ((Weight)Hidden[h].inputWeights.elementAt(0)).weight ;
    y_weight = ((Weight)Hidden[h].inputWeights.elementAt(1)).weight ;
    excitation = xy[0]*x_weight + xy[1]*y_weight + Hidden[h].biasWeight ;

    if ((excitation > -kconstant) && (excitation < kconstant)) {
         xterm += (((Weight)Hidden[h].outputWeights.elementAt(0)).weight *
               (x_weight / ( kconstant*2.0 ))) ;
         yterm += (((Weight)Hidden[h].outputWeights.elementAt(0)).weight *
               (y_weight / ( kconstant*2.0 ))) ;
         cterm += (((Weight)Hidden[h].outputWeights.elementAt(0)).weight *
               ((Hidden[h].biasWeight / ( kconstant * 2.0 ))+0.5 )) ;
    }

    if ( excitation >= kconstant )
         cterm += ((Weight)Hidden[h].outputWeights.elementAt(0)).weight ;
     }

     double m = -xterm / yterm ;
     double c = -cterm / yterm ;
```

```
 for( vertex = 0 ; vertex < region.length–1 ; vertex += 2 ) {
x1=region[vertex] ;
y1=region[vertex+1] ;
if ( vertex >= region.length–2 ) {
     x2=region[0] ;
     y2=region[1] ;
} else {
     x2=region[vertex+2];
     y2=region[vertex+3];
}

//If we detect an intersection, begin/end splitting the polygon
p1above = ( (m∗x1+c) < y1 );
p2above = ( (m∗x2+c) < y2 );
if ((p1above != p2above || (m∗x2+c) == y2) && m∗x1+c != y1 ) {

     if (x1==x2 && y1==y2) {
    System.out.println("Error :line of zero length");
     } else {
    // If represent gradient for both vertical and horizontal lines :
    if ( Math.abs( y2 – y1 ) < Math.abs( x2 – x1 ) || (y2–y1) ==
     (double)0 ) {
         grad = (y2–y1) / (x2–x1) ;
         ix = ( y1–grad ∗ x1 – c) / ( m – grad );
         iy = ( m ∗ ix ) + c ;
    } else {
         grad = (x2–x1) / (y2–y1);
         ix = (x1+grad∗c–grad∗y1) / (1–grad∗m);
         iy = (m ∗ ix) + c;
    }
    i[intersect++] = ix ;
    i[intersect++] = iy ;

     }
}
 }

 if ( intersect == 0 ) {
lines.addElement( new Double(0.0) ) ;
lines.addElement( new Double(0.0) ) ;
lines.addElement( new Double(0.0) ) ;
lines.addElement( new Double(0.0) ) ;
 } else if ( intersect == 4 ) {
for (vertex =0 ; vertex < intersect ; vertex++ )
```

```
        lines.addElement( new Double( i[vertex] ) ) ;
     } else {
    System.out.println("Error - wrong number of intersects") ;
     }
}
return (double[])rVecToArray( lines ) ;
 }


 public Object[] calculateRegions() {
// Initialise the first region on the list to the
// dimensions of the screen –an array of xy pairs :
Vector regions = new Vector() ;
double[] screen = new double[8] ;
screen[0] = IOSpaceCanvas.MINX ;
screen[1] = IOSpaceCanvas.MINY ;
screen[2] = IOSpaceCanvas.MINX ;
screen[3] = IOSpaceCanvas.MAXY ;
screen[4] = IOSpaceCanvas.MAXX ;
screen[5] = IOSpaceCanvas.MAXY ;
screen[6] = IOSpaceCanvas.MAXX ;
screen[7] = IOSpaceCanvas.MINY ;
regions.addElement( (Object)screen ) ;

       // Then split any region in the regions vector regions
// in two whenever it is intersected by a hyperband edge :
hBand[] bands = calculateHyperbands() ;
int x ;
for( x = 0; x < bands.length ; x++ ) {  // For each hyperband edge
     regions = splitRegions( regions, bands[x].gradient, bands[x].top ) ;
     regions = splitRegions( regions, bands[x].gradient, bands[x].bot ) ;
}

Object[] result = new Object[regions.size()] ;
regions.copyInto( result ) ;
return result ;
 }

 public Object[] calculateWintermute() {
// Initialise the first region on the list to the
// dimensions of the screen –an array of xy pairs :
Vector regions = new Vector() ;
int xpos ;
int ypos ;
```

```
int res = 50 ;
double width = ( IOSpaceCanvas.MAXX – IOSpaceCanvas.MINX ) / res ;
double height = ( IOSpaceCanvas.MAXY – IOSpaceCanvas.MINY ) / res ;
//System.out.println( "Width :"+width+" height "+height ) ;
for ( xpos = 0 ; xpos < res ; xpos ++ )
    for ( ypos = 0 ; ypos < res ; ypos++ ) {
   double[] region = new double[8] ;
   region[0] = IOSpaceCanvas.MINX+((double)xpos*width) ;
   region[1] = IOSpaceCanvas.MINY+((double)ypos*height) ;
   region[2] = IOSpaceCanvas.MINX+((double)xpos*width) ;
   region[3] = IOSpaceCanvas.MINY+((double)(ypos+1)*height) ;
   region[4] = IOSpaceCanvas.MINX+((double)(xpos+1)*width);
   region[5] = IOSpaceCanvas.MINY+((double)(ypos+1)*height);
   region[6] = IOSpaceCanvas.MINX+((double)(xpos+1)*width);
   region[7] = IOSpaceCanvas.MINY+((double)ypos*height) ;
   regions.addElement( (Object)region ) ;
    }
Object[] result = new Object[regions.size()] ;
regions.copyInto( result ) ;
return result ;
 }

 // Generates a list of plane objects      :
 // Each plane is represented as a double array,
 //    element 0 – Plane depth in Z for depth sorting
 //            1 – xyz
 //            4 ... More xyz coordinate pairs as needed
 public Object[] calculatePlanes() {
Object[] subregions = calculateRegions() ;
Object[] result = new Object[subregions.length] ;
int vertice, point, counter ;
double[] plane, region, center = new double[2] ;

// Convert each 2d region into a to 3d plane with depth index
// It does this by adding a z coordinate for each xy pair. The z is
// the y from the 2d region and the new y coordinate is the activation
// value (height) for the now xz point in the network. This makes the
// initial translation of the planes easy as z is depth into the screen.
// The first element of the 3d plane array will be used later when
// sorting the planes for screen depth allowing hidden planes to be painted
// over at plot time :
for ( counter = 0 ; counter < subregions.length ; counter++ ) {
    region = (double[])subregions[counter] ;
    // The plane – adding space for z coordinates and z depth sort value
```

```
      plane = new double[(region.length/2)*3] ;
      point = 0 ;
      for ( vertice = 0 ; vertice < plane.length ; vertice++ )
     if ( (vertice % 3) == 1 ) {
          center[0] = region[point-1] ;
          center[1] = region[point] ;
          plane[vertice] = regionActivation( center ) ;
     } else {
          plane[vertice] = region[point++] ;
     }
      result[counter] = (Object)plane ;
}
return result ;
 }

 public double regionActivation( double[] xy ) {
double x_weight, y_weight, bias_weight, excitation = 0.0 ;
double out = Output.biasWeight ;
int h ;
for (h=0; h<Hidden.length; h++) {
     x_weight = ((Weight)Hidden[h].inputWeights.elementAt(0)).weight ;
     y_weight = ((Weight)Hidden[h].inputWeights.elementAt(1)).weight ;
     excitation = xy[0]*x_weight + xy[1]*y_weight + Hidden[h].biasWeight ;

     if ((excitation > -kconstant) && (excitation < kconstant))
    out += (excitation/(2*kconstant)+0.5)*
          ((Weight)Hidden[h].outputWeights.elementAt(0)).weight ;
     if (excitation >= kconstant)
    out += ((Weight)Hidden[h].outputWeights.elementAt(0)).weight ;
}

// Find the activation of the output unit, and return this result :
if ( (out > -kconstant) && (out < kconstant)    )
     out = out / (2*kconstant) + 0.5 ;
else if ( out <= -kconstant )
     out = 0.0 ;
else if ( out >= kconstant )
     out = 1.0 ;

    return out ;
 }

 public int regionSide( double[] xy ) {
int h, result = 0 ;
```

```
double x_weight, y_weight, excitation = 0.0 ;

for (h=0; h<Hidden.length; h++) {
    x_weight = ((Weight)Hidden[h].inputWeights.elementAt(0)).weight ;
    y_weight = ((Weight)Hidden[h].inputWeights.elementAt(1)).weight ;
    excitation = xy[0]*x_weight + xy[1]*y_weight + Hidden[h].biasWeight ;

    if ((excitation > -kconstant) && (excitation < kconstant))
   result = 2 ;

    if ( excitation >= kconstant     && result != 2)
   result = 1 ;
}

return result ;
 }

 public Vector splitRegions( Vector regions, double hbm, double hbc ) {
int orignum = regions.size() ;
int vertex, rgn ;
double x1, y1, x2, y2, ix, iy, grad ;
boolean p1above, p2above ;

double[] region ;
Vector subregionA, subregionB ;
boolean intersect = false ;
Vector result = new Vector() ;

// Go through each region in turn, and decide whether it is
// intersected in two places by this hyperband. If so split
// the region into two regions :
for ( rgn = 0 ; rgn < regions.size() ; rgn++ )
    if ( ((double[])regions.elementAt(rgn)).length > 0 )          {

   region = (double[])regions.elementAt(rgn) ;

   // Find out the number of intersections between this
   // hyperband line, and the region given. Keep intermediate
   // results from this calculation to help split the region :
   subregionA = new Vector() ;
   subregionB = new Vector() ;
   intersect = false ;
   for( vertex = 0 ; vertex < region.length-1 ; vertex += 2 ) {
        x1=region[vertex] ;
```

```
 y1=region[vertex+1] ;
 if ( vertex >= region.length–2 ) {
x2=region[0] ;
y2=region[1] ;
 } else {
x2=region[vertex+2];
y2=region[vertex+3];
 }

 // If we detect an intersection, begin/end splitting the polygon
 p1above = ( (hbm∗x1+hbc) < y1 );
 p2above = ( (hbm∗x2+hbc) < y2 );
 if ((p1above != p2above || (hbm∗x2+hbc) == y2) && hbm∗x1+hbc != y1 )
  {

if (x1==x2 && y1==y2) {
    System.out.println("Error :line of zero length");
} else {

    // If represent gradient for both vertical and horizontal lines :
    if ( Math.abs( y2 – y1 ) < Math.abs( x2 – x1 ) || (y2–y1) ==
     (double)0 ) {
   grad = (y2–y1) / (x2–x1) ;
   ix = ( y1–grad ∗ x1 – hbc) / ( hbm – grad );
   iy = ( hbm ∗ ix) + hbc ;
   //System.out.println("horizontal ix = "+ix+" iy ="+iy) ;
    } else {
   grad = (x2–x1) / (y2–y1);
   ix = (x1+grad∗hbc–grad∗y1) / (1–grad∗hbm);
   iy = (hbm ∗ ix) + hbc;
   //System.out.println("vertical ix = "+ix+" iy ="+iy) ;
    }

    // We only deal with two intersections of the polygon
    // so either start subregion b or close it for good :
    if ( subregionB.size() == 0 || intersect ) {
   intersect = !intersect ;
   subregionA.addElement( new Double( ix ) ) ;
   subregionA.addElement( new Double( iy ) ) ;
   subregionB.addElement( new Double( ix ) ) ;
   subregionB.addElement( new Double( iy ) ) ;
    }
}
 }
```

```
        // Add the x and y vertices of the current edge to subregion a or b
        if ( !intersect ) {
       subregionA.addElement( new Double( x2 ) ) ;
       subregionA.addElement( new Double( y2 ) ) ;
        } else {
       subregionB.addElement( new Double( x2 ) ) ;
       subregionB.addElement( new Double( y2 ) ) ;
        }
    }

    // If two sub regions were generated from this region, remove the
    // origional region from the vector, and add the two sub regions :
    if ( subregionB.size() > 0 && !intersect ) {
      result.addElement( rVecToArray( subregionA ) ) ;
      result.addElement( rVecToArray( subregionB ) ) ;
    } else {
      result.addElement( region ) ;
    }
     }
return result ;
 }

 // Given a vector of Double container objects, this returns an
 // a double[] array object containing primitive double elements :
 private Object rVecToArray( Vector region ) {

double[] result = new double[region.size()] ;
int counter ;
for ( counter = 0 ; counter < result.length ; counter++ )
     result[counter] = ((Double)region.elementAt( counter )).doubleValue() ;

return (Object)result ;
 }

 // Returns an array of hyperband objects, one for each hidden layer neuron :
 public hBand[] calculateHyperbands() {
hBand[] result = new hBand[Hidden.length] ;
int counter ;
for ( counter = 0 ; counter<Hidden.length ; counter++ )
     result[counter] = Hidden[counter].findHyperBand( bias, kconstant ) ;
return result ;
 }
```

```
    public static double[] regionCentre( double[] region ) {

   double[] result = new double[2] ;
   result[0] = 0.0 ;
   result[1] = 0.0 ;
   int counter, points = 0 ;
      for ( counter = 0 ; counter < (region.length–1) ; counter+=2 ) {
       result[0] += region[counter] ;
       result[1] += region[counter+1] ;
       points++ ;
   }
   result[0] = result[0] / (double)(points) ;
   result[1] = result[1] / (double)(points) ;
   return result ;
    }
}

// Neuron class used to represent one neuron unit of a strictly layered
// feed forward artificial neural network. A neuron object contains all
// weights and pointers to the connecting input and output neuron(s).
class Neuron {

  public Vector outputWeights = new Vector() ;
  public Vector inputWeights = new Vector() ;
  public double biasWeight = 0 ;

    // Initialises the neuron data structures and connects weights
    // between this neuron, and all of the neurons providing input
    // to it :
    public Neuron( Neuron[] inputs, Random generator ) {
   // Generate a new weight connections between this neuron and its inputs
   if ( inputs != null ) {
       int counter ;
       for ( counter = 0 ; counter < inputs.length ; counter ++ )
      inputWeights.addElement( new Weight( inputs[counter], this, generator ) ) ;
       biasWeight = 1–(generator.nextDouble()*2) ;
   }
    }

    public Neuron() {
   biasWeight = 0.01 ;
    }
```

```
    public void addWeight( Neuron toNeuron, double value ) {
    Weight toAdd = new Weight( this, toNeuron, value ) ;
    //System.out.println("Trying to add between "+this+" and "+toNeuron+" "
     +value ) ;
    toNeuron.inputWeights.addElement( toAdd ) ;
     }

     // Function to generate a hyperband, given that this is a hidden layer neuron
     // to be written...
     public hBand findHyperBand( double bias, double k ) {
    hBand result = new hBand() ;
    if ( inputWeights.size() != 2 ) {
        System.out.println("This is not a two input hidden layer neuron!") ;
        result = null ;
    } else {
        // X is assumed to be the first input and y the second :
        double w1x, w1y, wbias ;
        w1x = ((Weight)inputWeights.elementAt(0)).weight ;
        w1y = ((Weight)inputWeights.elementAt(1)).weight ;
        result.gradient = ( -w1x / w1y ) ;
        result.top = (k/w1y) - (biasWeight/w1y)    ;
        result.bot = (-k/w1y) - (biasWeight/w1y) ;
    }
    return result ;
     }

}

// Represents the weight value that connects two neurons
class Weight {

     public Neuron from ;
     public Neuron to ;
     public double weight ;

     public Weight( Neuron efferent, Neuron afferent, double value ) {
    from = efferent ;
    to = afferent ;
    weight = value ;
    from.outputWeights.addElement( this ) ;
     }

     public Weight( Neuron efferent, Neuron afferent, Random generator ) {
```

```
    // Initialise all values pertaining to the weight object :
    from = efferent ;
    to = afferent ;
    weight = 1-(generator.nextDouble()*2) ;

    // The input neuron to this weight line has the weight added to it :
    from.outputWeights.addElement( this ) ;
     }

}
```

## 12.4 The hyperband data storage module hBand.java

```
/*****************************************
 *       Hyperband data structure module                 *
 * By Duncan McPherson                    *
 *                                        *
 *    This module is used to represent a hyperband        *
 *****************************************/

class hBand {
     public double gradient = 1.0 ;
     public double width = 20.0 ;
     public double top = 0.0 ;
     public double bot = 0.0 ;
}
```

## 12.5 The input output display module IOSpaceCanvas.java

```
/*****************************************\
 *            JSand Display Module
 * By Duncan McPherson
 *
 * The display canvas module is stored in this source
 * and is common to the applet and   java application
\*****************************************/

import java.awt.event.*;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.awt.image.*;
```

```
import java.lang.Math.*;


/*********************************************\
 * Class : IOSpaceCanvas
 *
 * Purpose : Draws the Input/Output space of a neural network
 *
\*********************************************/
class IOSpaceCanvas extends Canvas    {

    hBand[] bands = null ;
    double[] boundaries = null ;
    Object[] regions = null ;
    Object[] planes = null ;
    Object[] translated = null ;
    ANN network ;
    double zoom = 1; // scaling of the IOSpace canvas
    public static final double MINX = -25 ;
    public static final double MINY = -25 ;
    public static final double MAXX = 25 ;
    public static final double MAXY = 25 ;
    public static final double MAXZ = 20 ;

    // Boolean properties of the display - that may be turned on or off
    public boolean draw_bands = false ;
    public boolean draw_regions = true ;
    public boolean number_regions = false ;
    public boolean draw_axes = false ;
    public boolean boundary_lines = false ;
    public boolean step = false ;
    public boolean region_activation = false ;
    public int stnum = 0 ;
    int width, height ;
    double scale    ;

    // Specific to the 3d projection funcitonality - angles in degrees
    public boolean threed = false ;
    public int xrot = 0 ;
    public int yrot = 90 ;
    private boolean drag = false ;
    private int dragx = 0 ;
    private int dragy = 0 ;
    private int dragxrot = 0 ;
```

```
 private int dragyrot = 0 ;

 // Image to be used in double buffered redraws
 Image offImage ;

 public void paint( Graphics window ) {
update( window ) ;
 }

 public void update( Graphics window ) {
int counter ;
boolean fill ;
double[] region_centre ;
Dimension current_size = getSize();
width = current_size.width ;
height = current_size.height ;
scale = (height/(MAXX-MINX))*zoom ;

// If there is 3d data which should be plotted ensure it is translated
if ( threed && translated == null && planes != null )
    translated = translate_planes( xrot, yrot, planes ) ;

// Set up a graphics buffer to draw the new image into :
if ( offImage == null )
    offImage = createImage( width, height );
if ( offImage.getWidth( this ) != width || offImage.getHeight( this ) != height )
    offImage = createImage( width, height );
Graphics g = offImage.getGraphics();

// Colour the background sky blue for a full repaint
g.setColor( Color.cyan ) ;
g.fillRect( 0 , 0 , current_size.width, current_size.height ) ;

// Then plot the canvas according to the selected boolean switches and
// available data - if no plot data is generated nothing can be plotted :
if ( threed && translated != null ) {
    for ( counter = 0 ; counter < translated.length ; counter++ )
   if ( counter == stnum || !step)
        draw_plane( (double[])translated[counter], g, true ) ;
   else
        draw_plane( (double[])translated[counter], g, false ) ;

} else if ( step && regions != null ) {
    region_centre = ANN.regionCentre( (double[])regions[stnum] ) ;
```

```
      draw_region( (double[])regions[stnum], region_centre, g ) ;
      draw_a_boundary( g, stnum ) ;
      mark_point( region_centre[0], region_centre[1], 8, g ) ;
      label( new String( ""+(stnum+1) ), region_centre[0], region_centre[1], g ) ;

  } else if ( draw_regions && regions != null ) {
      for ( counter = 0 ; counter < regions.length ; counter++ ) {
     region_centre = ANN.regionCentre( (double[])regions[counter] ) ;
     draw_region( (double[])regions[counter], region_centre, g ) ;
     if (number_regions && !region_activation )
          label( new String( ""+(counter+1) ), region_centre[0],
            region_centre[1], g ) ;
      }
  }

  if ( draw_bands && bands != null && !threed )
      for ( counter = 0 ; counter < bands.length ; counter++ )
     draw_hyperband( bands[counter], g ) ;
  if ( boundary_lines && boundaries != null && !threed ) draw_boundaries( g ) ;
  if ( draw_axes && !threed ) draw_axes( g, 5.0 ) ;

  // And plot the entire buffered impage to the screen in one go
  window.drawImage(offImage, 0, 0, this);

   }

   public void drag( int x, int y ) {
  if ( drag == false && threed ) {
      drag = true ;
      dragx = x ;
      dragy = y ;
      dragxrot = xrot ;
      dragyrot = yrot ;
  } else if ( threed ) {
      xrot = (dragxrot+(y-dragy)) % 360 ;
      if ( xrot < 0 ) xrot += 360 ;
      yrot = (dragyrot+(x-dragx)) % 360 ;
      if ( yrot < 0 ) yrot += 360 ;
      translated = null ;
      repaint() ;
  }
   }

   public void drag_finished( int x, int y ) {
```

```
if ( drag == true ) {
     drag = false ;
     xrot = (dragxrot+(y–dragy)) % 360 ;
     if ( xrot < 0 ) xrot += 360 ;
     yrot = (dragyrot+(x–dragx)) % 360 ;
     if ( yrot < 0 ) yrot += 360 ;
     translated = null ;
     repaint() ;
}
 }

 private void mark_point( double xp, double yp, int size, Graphics g ) {
double scale = (height/(MAXX–MINX))∗zoom ;
int x = (width/2)+(int)(xp∗scale) ;
int y = (height/2)–(int)(yp∗scale) ;

g.setColor(Color.red);
g.fillOval(x–(int)(size/2),y–(int)(size/2),size,size);
g.setColor(Color.blue);
g.drawOval(x–(int)(size/2),y–(int)(size/2),size,size);
 }

 private void draw_hyperband( hBand hyperband, Graphics g ) {

// Scale the display a constant screen size multiplied by a zoom factor
// calulated by the (actual display::virtual display width width) ∗ zoom
double y1, y2, scale = (height/(MAXX–MINX))∗zoom ;
double x1 = MINX∗3 ; // Taking x and y coordinates at the edges of the
double x2 = MAXX∗3 ; // virtual display.

// Compute two y coordinates on the centre line of the hyperband :
y1 = (x1 ∗ hyperband.gradient) ;
y2 = (x2 ∗ hyperband.gradient) ;

// Draw the top and bottom border lines of the hyperband on the
// display with the display center being the origin of I/O space :
g.setColor( Color.blue ) ;
g.drawLine( (width/2)+(int)(x1∗scale), (height/2)–(int)((y1+hyperband.top)∗scale),
        (width/2)+(int)(x2∗scale), (height/2)–(int)((y2+hyperband.top)∗scale) ) ;
g.drawLine( (width/2)+(int)(x1∗scale), (height/2)–(int)((y1+hyperband.bot)∗scale),
        (width/2)+(int)(x2∗scale), (height/2)–(int)((y2+hyperband.bot)∗scale) ) ;
 }

 private void draw_region( double[] region, double[] region_centre, Graphics g ) {
```

```
double scale = (height/(MAXX–MINX))∗zoom ;
Polygon p = new Polygon() ;
int counter ;
for ( counter = 0 ; counter < (region.length–1) ; counter+=2 )
    p.addPoint( (width/2)+(int)(region[counter]∗scale),
      (height/2)–(int)(region[counter+1]∗scale) ) ;

if ( region_activation ) {
    int rgb = (int)(network.regionActivation( region_centre ) ∗ 400) ;
    if ( rgb < 0 ) rgb = 0 ;
    if ( rgb > 255 ) rgb = 255 ;
    g.setColor( new Color( rgb, rgb, rgb ) ) ;
} else {
    int on = network.regionSide( region_centre ) ;
    // Draw the region in red, and the outline in black :
    if ( on == 0 )
   g.setColor( Color.orange ) ;
    else if ( on == 1 )
   g.setColor( Color.yellow ) ;
    else
   g.setColor( Color.white ) ;
}

g.fillPolygon(p) ;
 }

 private void draw_plane( double[] plane, Graphics g, boolean fill ) {

double scale = (height/(MAXX–MINX))∗zoom ;
Polygon p = new Polygon() ;
int counter ;
for ( counter = 1 ; counter < (plane.length–1) ; counter+=2 )
    p.addPoint( (width/2)+(int)(plane[counter]∗scale),
      (height/2)–(int)(plane[counter+1]∗scale) ) ;
int rgb = (int)(plane[0] ∗ 400);
if ( rgb < 0 ) rgb = 0 ;
if ( rgb > 255 ) rgb = 255 ;
g.setColor( new Color( rgb, rgb, rgb ) ) ;
if ( fill ) g.fillPolygon(p) ;
g.setColor( Color.blue ) ;
if ( draw_bands || step ) g.drawPolygon(p) ;
 }
```

```
 private Object[] translate_planes( int x, int y, Object[] planes ) {
double A, B, C, D, M, N, X, Y, Z, yheight ;
int counter, vertice, point ;
double[] plane, region, depthsort = new double[planes.length] ;
Object[] result = new Object[planes.length] ;

// Find the trigonometric terms used to translate about x and y axes
A = Math.cos((Math.PI/180)*(double)x);
B = Math.sin((Math.PI/180)*(double)x);
C = Math.cos((Math.PI/180)*(double)y);
D = Math.sin((Math.PI/180)*(double)y);

// Then translate all xyz vertice coordinates
for ( counter = 0 ; counter < planes.length ; counter++ ) {
     plane = (double[])planes[counter] ;
     region = new double[(plane.length/3)*2 +1] ;
     yheight = 0.0 ;
     point = 1 ;
     vertice = 0 ;
     while (point < region.length ) {
    X=plane[vertice++] ;
    Y=plane[vertice++] ;
    Z=plane[vertice++] ;
    yheight += Y ;
    // Scale the altitude of the plane, and rotate about x and y axes
    Y = (0.5–Y)*(MAXZ) ;
    M = Y * A – Z * B ;
    N = Z * A + Y * B ;
    Z = M ;
    Y = N ;
    M = X * C – Z * D ;
    N = Z * C + X * D ;
    region[point++] = N ;
    region[point++] = Y ;
    if (depthsort[counter]<M) depthsort[counter]=M ; // And finally the depth
     index
     }
     // For colouring this plane average the untranslated y height
     region[0] = yheight / (double)(plane.length/3) ;
     result[counter] = region ;
}

// Finally sort the planes by their depth z coordinate for plotting
int i ;
```

```
Object temp ;
double tempindex ;
for ( counter = 1 ; counter < depthsort.length ; counter++ ) {
     i = counter ;
     while ( i > 0 && depthsort[i] < depthsort[i–1] ) {
    tempindex = depthsort[i] ;
    temp = result[i] ;
    result[i] = result[i–1] ;
    depthsort[i] = depthsort[i–1] ;
    result[i–1] = temp ;
    depthsort[i–1] = tempindex ;
    i –– ;
     }
}

return result ;
 }

 private void draw_boundaries( Graphics g ) {
double scale = (height/(MAXX–MINX))∗zoom ;
g.setColor( Color.red ) ;
int counter ;
if (boundaries.length > 3)
     for ( counter = 0 ; counter < boundaries.length–3 ; counter+=4 )
    g.drawLine( (width/2)+(int)(boundaries[counter]∗scale),
              (height/2)–(int)(boundaries[counter+1]∗scale),
              (width/2)+(int)(boundaries[counter+2]∗scale),
              (height/2)–(int)(boundaries[counter+3]∗scale) ) ;
 }

 private void draw_a_boundary( Graphics g, int boundary ) {
double scale = (height/(MAXX–MINX))∗zoom ;
g.setColor( Color.red ) ;
int counter, num = 0 ;
if (boundaries.length > 3)
     for ( counter = 0 ; counter < boundaries.length–3 ; counter+=4 ) {
    if ( num == boundary )
         g.drawLine( (width/2)+(int)(boundaries[counter]∗scale),
           (height/2)–(int)(boundaries[counter+1]∗scale),
           (width/2)+(int)(boundaries[counter+2]∗scale),
           (height/2)–(int)(boundaries[counter+3]∗scale) ) ;
    num++ ;
     }
 }
```

```
    private void draw_axes( Graphics g, double inc ) {
   double scale = (height/(MAXX–MINX))∗zoom ;
   int counter ;

   g.setColor( Color.black ) ;
   g.drawLine( (width/2)+(int)(MINX∗scale), (height/2),
            (width/2)+(int)(MAXX∗scale), (height/2) ) ;
   g.drawLine( (width/2), (height/2)–(int)(MINY∗scale),
            (width/2), (height/2)–(int)(MAXY∗scale) ) ;

   double pos = 0.0 ;
   while ( pos < MAXX ) {
        pos += inc ;
        g.drawLine( (width/2)+(int)(pos∗scale), (height/2)+2,
          (width/2)+(int)(pos∗scale), (height/2)–2 ) ;
        g.drawLine( (width/2)+(int)((–pos)∗scale), (height/2)+2,
          (width/2)+(int)((–pos)∗scale), (height/2)–2 ) ;
   }
   pos = 0.0 ;
   while ( pos < MAXY ) {
        pos += inc ;
        g.drawLine( (width/2)–2, (height/2)–(int)(pos∗scale),
          (width/2)+2, (height/2)–(int)(pos∗scale) ) ;
        g.drawLine( (width/2)–2, (height/2)–(int)(–pos∗scale),
          (width/2)+2, (height/2)–(int)(–pos∗scale) ) ;
   }
    }

    // Add a text label to a region or area of the display :
    private void label( String label, double x, double y, Graphics g ) {
   double scale = (height/(MAXX–MINX))∗zoom ;
   Font f = new Font("Helvetica", Font.BOLD, 14 ) ;
   g.setFont( f ) ;
   g.setColor( Color.blue ) ;
   FontMetrics fm = g.getFontMetrics( f ) ;
   g.drawString(label, (width/2)+(int)(x∗scale) – (fm.stringWidth(label) /2),
            (height/2)–(int)(y∗scale) );
    }
}
```

## 12.6 The subgoal display applet SubgoalApplet.java

```
/*****************************************\
 *          SubGoal Chain Display Applet
 *
 *             By Duncan McPherson
 *            17th of February 1999
\*****************************************/

import java.applet.Applet;
import java.awt.Component;
import java.lang.*;
import java.math.* ;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.text.* ;


/*************************************************\
 * Class : SubGoalsApplet - Main applet class
 *
 * Purpose : The core applet routine which intialises the
 *         applet GUI, and begins communication with the
 *         server program. Once communication has begun a
 *         separate threaded object 'SubGoalReceiver' is used
 *         to allow the GUI to remain responsive whilst
 *         receiving new points from the ANN simulator.
\*************************************************/
public class SubgoalsApplet extends Applet {

    TextField extension;
    TextArea textbox;
    Button go, disconnect;

    SubGoalCanvas chainDisplay ;
    SubGoalReceiver receiver ;
    SubGoalChain data ;

    Label ext, stat ;
    private boolean laidOut = false;
```

```
 // Communications with server variables :
 DataInputStream is = null ;
 Socket s = null;
 PrintStream os = null ;

 public void init() {

setFont(new Font("Helvetica", Font.PLAIN, 14));
ext = new Label("Enter the server name here:");
      add( ext );
stat = new Label("Data from the ANN simulator:");
add( stat );
textbox = new TextArea(10,40);
add( textbox );
go = new Button("Start");
      add( go );
disconnect = new Button("Disconnect");
      add( disconnect );
extension = new TextField( 20 );
      extension.setEditable(true);
extension.setText("orkney.dcs") ;
add( extension ) ;

// Create a subgoal canvas and data storage object.
// At the start we are not connected to the server
// and this will not change until the user clicks
// the start button :
chainDisplay = new SubGoalCanvas() ;
add( chainDisplay ) ;
data = new SubGoalChain( 680, chainDisplay ) ;

      validate();
 }

 public void paint(Graphics g) {

// If not yet done, put all the required components
// in their places within the applet window :
      if (!laidOut) {
           Insets insets = insets();
           ext.reshape(10 + insets.left, 17 + insets.top, 200, 13);
           extension.reshape( 10 + insets.left, 30 + insets.top, 200, 42);
           go.reshape(220 + insets.left,      30 + insets.top, 80, 42 );
```

```
        disconnect.reshape(320 + insets.left, 30 + insets.top, 80, 40);
    stat.reshape(10 + insets.left, 97 + insets.top, 300, 13 );
        textbox.reshape(10 + insets.left, 110 + insets.top, 400, 150 );
    chainDisplay.reshape( 10+insets.left, 280+insets.top, 780, 300 ) ;
        laidOut = true;
}
 }

 public boolean action(Event e, Object arg){

if ( e.target == disconnect ){
    // Disconnect from the server here :
    disconnect() ;
    return true;
}  else if ( e.target == go ) {
    if ( receiver != null ) {
   receiver.readThread.stop() ;
   if ( s != null ) disconnect() ;
    }
    String host = extension.getText();
    textbox.appendText("Try to connect to host "+host+"\n" ) ;

    // Attempt to connect to the given server name and then
    // involk the ANN simulator :
    try {
   s = new Socket(host, 5555 ) ;
   is = new DataInputStream(s.getInputStream() ) ;
   os = new PrintStream( s.getOutputStream() ) ;
   os.print("ANNSIMSERVER") ;
   textbox.appendText("Server initialised\n") ;
   data = new SubGoalChain( 680, chainDisplay ) ;
   // Create a new thread which handles all
   // output from the newly connected server :
   receiver = new SubGoalReceiver( this, is ) ;
    } catch(Exception ex) {
   textbox.appendText("Unable to connect to "+host+"\n") ;
   return false ;
    }

    return true;
}
return false ;
 }
```

```
    public void disconnect() {
   if ( s != null )
       try {
      s.close() ;
       } catch( Exception ex ) {}
   s = null ;
    }
}


/*********************************************\
 * Class : SubGoalReceiver
 *
 * Purpose : Manages all I/O between Applet and ANN Simulator
 *           which is running remotely. Communication via BSD
 *           sockets. This class executes as a standalone
 *           thread in parallel with the rest of the applet
\*********************************************/
class SubGoalReceiver implements Runnable
{
    private DataInputStream inputStream; // Holds the input byte stream from
     reciever
    Thread readThread; // This thread, which must be active to recieve bytes
    SubgoalsApplet display ;

    // Tokens in the input stream which are parsed in order
    // to extract output states being generated by the ANN
    // simulator program.
    private static final String SUBGOAL = "SG_" ;
    private static final String GOAL = "O_g_0 =" ;
    private static final String WEIGHT = "w_c_0 =" ;
    private static final String OUTPUT = "O_c_0 =" ;
    private static final String DONE = "Done" ;

    public SubGoalReceiver( SubgoalsApplet thedisplay, DataInputStream input ) {
   // Copy the instance of the reciever object, and set up buffer
   // variables and flags associated with decoding incoming frames ...
   display = thedisplay ;
   inputStream = input ;
   readThread = new Thread(this);
   readThread.start();
    }

    // Entry point for the thread on start of execution :
```

```
 public void run() {
      try {
           Thread.sleep(1000);
      } catch (InterruptedException e) {}

String temp = null;
double[] val = null;
int lastGoal = -1 ;
boolean goal = false ;
boolean finished = false ;
int iterations = 0 ;

while ( !finished ) {
     try {
    temp = inputStream.readLine() ;
     } catch(Exception ex) { System.out.println("Error" + ex) ; }

     // Parse the input stream line by line for known tokens :
     if ( temp != null ) {
    temp = temp.trim() ;

    // A new subgoal is found in the input stream
    if ( temp.startsWith( SUBGOAL ) ) {
         temp = temp.substring( SUBGOAL.length(), temp.length() ) ;

         int sgnum = Integer.parseInt(temp.substring( 0, temp.indexOf("=") ).trim(),
          10);
         val = parseFloats( temp.substring( temp.indexOf("=")+1,
                        temp.length()).trim() ) ;
         int counter ;
         temp = "SubGoal "+sgnum+" : " ;
         for ( counter = 0 ; counter < val.length ; counter++ )
        if ( counter == (val.length-1) )
             temp += val[counter] ;
        else
             temp += val[counter] + " , " ;
         display.textbox.appendText( temp+"\n" );
         display.data.addSubGoal( val , sgnum ) ;
    }

    // A new output state is found in the input stream
    else if ( temp.startsWith(OUTPUT) ) {

         iterations++ ;
```

```
            val = parseFloats( temp.substring( temp.indexOf("=")+1,
                                temp.length()).trim() ) ;
            int counter ;
            temp = "Output : " ;
            for ( counter = 0 ; counter < val.length ; counter++ )
           if ( counter == (val.length–1) )
                temp += val[counter] ;
           else
                temp += val[counter] + " , " ;
            display.textbox.appendText( temp+"\n" );
            display.data.addOutput( new ndPoint( val ) ) ;
        }

        // A goal state is found in the input stream only register
        // this once, since the goal will be the same throughout
        // the training period we are displaying
        else if ( temp.startsWith(GOAL) && goal == false ) {
            goal = true ;
            val = parseFloats( temp.substring( temp.indexOf("=")+1,
                                temp.length()).trim() ) ;
            int counter ;
            temp = "Goal : " ;
            for ( counter = 0 ; counter < val.length ; counter++ )
           if ( counter == (val.length–1) )
                temp += val[counter] ;
           else
                temp += val[counter] + " , " ;
            display.textbox.appendText( temp+"\n" );
            display.data.setGoal( new ndPoint( val ) ) ;
        }

        // The 'Done' statement is found in the input stream
        else if ( temp.startsWith( DONE ) ) {
            display.textbox.appendText( "Finished training after "+
                    iterations+" iterations.");
            finished = true ;
        }
         }
    }

    try {
         inputStream.close() ;
    } catch (IOException e) {}
```

```
    // Unhook the socket connection between the display and the server :
    display.data.update() ;
    display.disconnect() ;
     }

     // Given a bracketed list of comma separated floating point numbers in
     // a string, this function returns an array containing the floats :
     private double[] parseFloats( String flist ) {

    // Strip the brackets around the set of floating point numbers :
    if ( flist.startsWith("(") ) flist = flist.substring( 0, flist.length() ) ;
    if ( flist.endsWith(")") ) flist = flist.substring( 1, flist.length() -1 ) ;

    // Count how many numbers - by separating commas are within the text
    String number ;
    int counter, comma = 1 ;
    for ( counter = 0 ; counter < flist.length() ; counter++ )
         if (flist.substring(counter, counter+1).equals(",") )
        comma++ ;
    double[] result = new double[comma] ;

    // Then extract the floating point values from the text list :
    comma = 0 ;
    int value = 0 ;
    for ( counter = 0 ; counter < flist.length() ; counter++ )
         if (flist.substring(counter, counter+1).equals(",") ) {
        try {
             result[value] = Float.valueOf( flist.substring(comma,
                                  counter ).trim()).floatValue()        ;
        } catch ( Exception e ) { result[value] = (double)0.0 ;}
        value++ ;
        comma = counter+1 ;
         }

    result[value] = Float.valueOf(flist.substring( comma, counter ).trim()).floatValue()
     ;
    return result ;
     }
}

/*********************************************\
 * Class : SubGoalCanvas
 *
 * Purpose : Creates the subgoal display canvas which is used
```

```
 *          to form part of the complete applet GUI interface.
 \*****************************************************/
class SubGoalCanvas extends Canvas {

    // Private storage of the subgoal data to be displayed
    public xyPoint chain[] = null ;
    public xyPoint goals[] = null ;

    public boolean showgoals = true ;
    public boolean showattained = true ;
    public int oldwidth = 0 ;
    public boolean new_point = false;

    // Image to be used in double buffered redraws
    Image offImage ;
    public boolean dbuffer = true ;


    // Main redrawing procedure
    public void paint( Graphics window ) {
   update( window ) ;
    }

    public void update( Graphics window ) {
   // Then try to draw the subgoal display with what data we
   // have at present :

   Dimension current_size = getSize();
   Graphics g ;
   String s1 = "goal" ;
   String s2 = "initial" ;

   int xoffset = 50 ; // Border width at either side of display
   int bottomy = 0; // space at bottom of screen for text
   int yoffset = (current_size.height-bottomy) /2;
   int width = current_size.width - (2*xoffset);
   int link, oldx, oldy ;

   // Set up a graphics buffer to draw the new image into :
   if ( offImage == null )
        offImage = createImage( current_size.width, current_size.height );
   if ( offImage.getWidth( this ) != current_size.width ||
         offImage.getHeight( this ) != current_size.height )
        offImage = createImage( current_size.width, current_size.height );
```

```
// If the flag is set to perform double buffering then draw into the buffer :
if ( dbuffer )
     g = offImage.getGraphics();
else
     g = window ;

// Block out the background in white first :
g.setColor( Color.white ) ;
g.fillRect( 0 , 0 , current_size.width, current_size.height ) ;

// If the user has selected marking of subgoals, then
// draw them too in the appropriate positions on screen :
if ( goals != null     ) {

     g.setColor(Color.green);
     g.drawLine(goals[0].x + xoffset ,goals[0].y + yoffset ,
             goals[goals.length-1].x + xoffset ,goals[goals.length-1].y + yoffset);

     g.setColor(Color.blue);

     label_subgoal( "initial", goals[0].x+xoffset,
           goals[0].y+yoffset,8,g ) ;

     label_subgoal( "goal", goals[goals.length-1].x+xoffset,
           goals[goals.length-1].y+yoffset,8,g ) ;

     if (showgoals == true) {
    for ( link = 0 ; link < goals.length ; link++ ) {
         mark_subgoal( goals[link].x + xoffset ,
              goals[link].y + yoffset, 8, g ) ;
         //label_subgoal( new String(""+link+""), goals[link].x+xoffset,
         //              goals[link].y+yoffset,8,g ) ;
    }
     } else {
    mark_subgoal( goals[0].x + xoffset ,
               goals[0].y + yoffset, 8, g ) ;

    mark_subgoal( goals[goals.length-1].x + xoffset ,
               goals[goals.length-1].y + yoffset, 8, g ) ;
     }
}

// If a subgoal chain has been sucessfully
```

```
// generated, then display it :
g.setColor(Color.red);

if ( chain != null    )  {
     oldx = 0 ;
     oldy = 0 ;

     for ( link = 0 ; link < chain.length ; link++ ) {
    g.drawLine( oldx + xoffset, oldy + yoffset,
              chain[link].x + xoffset ,
              chain[link].y + yoffset ) ;

    oldx = chain[link].x ;
    oldy = chain[link].y ;
     }

     if (showattained == true)
    for ( link = 0 ; link < chain.length ; link++ )
         mark_point(chain[link].x+xoffset, chain[link].y+yoffset,4,g);
}

// And plot the entire buffered image to the screen in one go
if ( dbuffer )
     window.drawImage(offImage, 0, 0, this);
 }

 // Procedure used to draw a point in the subgoal display :
 private void mark_point( int x, int y, int size, Graphics g ) {
g.setColor(Color.red);
g.fillOval(x–(int)(size/2),y–(int)(size/2),size,size);
g.setColor(Color.blue);
g.drawOval(x–(int)(size/2),y–(int)(size/2),size,size);
 }

 // Marks a subgoal item with a cross onscreen :
 private void mark_subgoal( int x, int y, int size, Graphics g ) {
g.setColor(Color.yellow);
g.fillOval(x–(int)(size/2),y–(int)(size/2),size,size);
g.setColor(Color.red);
g.drawOval(x–(int)(size/2),y–(int)(size/2),size,size);
 }

 // Add a text label to a subgoal in the display :
 private void label_subgoal( String label, int x, int y,
```

```
            int size, Graphics g ) {

    Font f = new Font("Helvetica", Font.BOLD, 14 ) ;
    g.setFont( f ) ;
    FontMetrics fm = g.getFontMetrics( f ) ;
    g.drawString(label, x – (fm.stringWidth(label) /2),
            y–(int)(size∗1.4) ) ;
     }
}

/∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗\
 ∗ Class : SubGoalChain
 ∗
 ∗ Purpose : Reads and manages the data input from the ANN
 ∗          simulator, and generates a subgoal chain using
 ∗          this input
\∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗/
class SubGoalChain {
     // All class member variables are initialised
     // to a blank canvas state. There is no need for
     // a constructor until a connection with the server
     // has been established.

     // Arrays used to hold nd and 2d representations
     ndPoint start_point = null ;
     Vector nd_outputs = new Vector() ;
     Vector nd_subgoals = new Vector() ;
     ndPoint goal_point = null ;

     boolean[] dimensions = null ;
     int display_width ;
     SubGoalCanvas display = null ;

     // A new subgoal chain display is created using three input nd points,
     // a first subgoal weight state, a current weight state and the current weight
      state
     public SubGoalChain( int width, SubGoalCanvas thedisplay ) {

    // Initialise the display module to use all of the n dimensions
    display_width = width ;
    display = thedisplay ;
     }

     // Given a new point from the simulator, adds this to the
```

```
 // subgoal chain as it stands.
 public void addOutput( ndPoint newPoint ) {
if ( start_point == null )
    start_point = newPoint ;
else
    nd_outputs.addElement( (Object)newPoint ) ;
display.chain = outputs() ;
display.repaint() ;
 }

 public void addSubGoal( double[] pos, int goalNumber ) {
if ( goalNumber != 0 ) {
    if ( nd_subgoals.size() <= goalNumber )
   nd_subgoals.addElement( (Object)new ndPoint( pos ) ) ;

    else
   ((ndPoint)nd_subgoals.elementAt( goalNumber )).position = pos ;
}
display.goals = goals() ;
display.repaint() ;
 }

 public void setGoal( ndPoint theGoal ) {
System.out.println("Setting the goal") ;
goal_point = theGoal ;
 }

 public void update() {
display.chain = outputs() ;
display.goals = goals() ;
display.repaint() ;
 }

 // Updates the display with new dimensions and width variables
 public xyPoint[] outputs() {

double magnitude, angle, display_scale, display_angle ;
xyPoint result[] = null ;

if ( nd_outputs.size() > 0 && start_point != null ) {

    // Display angle, and magnitudes are calculated according to
    // the magnitude between initial and final output states of
    // the network (the initial state, and the goal are then
```

```
    // assumed to be at opposite ends of the display window – initial
    // on the left and goal on the right side of the display )
    display_scale = (double)display_width /
   ndPoint.magnitude( goal_point.position,
              start_point.position) ;
    display_angle = ndPoint.angle( goal_point.position,
                 start_point.position ) ;

    // Generate a result array for xy coordinate pairs to be displayed :
    result = new xyPoint[nd_outputs.size()] ;

    int counter = 0 ;
    while ( counter < (result.length) ) {
   result[counter] =
       ((ndPoint)nd_outputs.elementAt(counter)).xy( start_point,
                        goal_point,
                        display_scale,
                        display_angle,
                        display_width ) ;
   counter++ ;
    }
}
return result ;
 }

 public xyPoint[] goals() {
double magnitude, angle, display_scale, display_angle ;

xyPoint result[] = null ;

if ( goal_point != null && start_point != null ) {

    // Display angle, and magnitudes are calculated according to
    // the magnitude between initial and final output states of
    // the network (the initial state, and the goal are then
    // assumed to be at opposite ends of the display window – initial
    // on the left and goal on the right side of the display )

    display_scale = (double)display_width /
   ndPoint.magnitude( goal_point.position, start_point.position ) ;
    display_angle = ndPoint.angle( goal_point.position, start_point.position ) ;

    // Coordinate array for the start point, all subgoals and, and the goal :
    result = new xyPoint[nd_subgoals.size()+1] ;
```

```
        result[0] = start_point.xy( start_point, goal_point, display_scale,
                  display_angle, display_width ) ;
        result[result.length–1] = goal_point.xy( start_point, goal_point, display_scale,
                        display_angle, display_width ) ;

        int counter = 1 ;
        while ( counter < (result.length–1) )
       result[counter++] =
           ((ndPoint)nd_subgoals.elementAt(counter–1)).xy( start_point,
                                goal_point,
                                display_scale,
                                display_angle,
                                display_width ) ;
    }
    return result ;
     }
}


/*********************************************\
 * Class : xyPoint
 *
 * Purpose : Used to represent an xy position on the output
 *         display. Generated from an ndPoint object.
\*********************************************/
class xyPoint { int x ; int y ; }


/*********************************************\
 *  Class : ndPoint
 *
 *  Purpose : Used to represent an n dimensional point and
 *          calcualte a lower dimensional represenation of
 *     that point as required for subgoal chain display.
\*********************************************/
class ndPoint {

     // The position of this point in nd space
     public double[] position ;

     public ndPoint( double[] pos ) {
    position = pos ;
     }
```

```
 // Given two arrays, each representing an n–dimensional point
 // in space, this routine returns the magnitude between the two :
 public static double magnitude( double[] vectorA, double[] vectorB ) {

double result = 0.0 ;
int index ;
for ( index = 0 ; index < vectorA.length ; index++ )
    result += ( vectorB[index] – vectorA[index] ) *
   ( vectorB[index] – vectorA[index] ) ;
return Math.sqrt( result ) ;
 }

 // Work out the magnitude of a single vector :
 public static double v_mag( double[] vector ) {

double result = 0.0 ;
int index ;
for ( index = 0 ; index < vector.length ; index++ )
    result += vector[index] * vector[index] ;
return Math.sqrt( result ) ;
 }

 // Given two arrays, each representing an n–dimensional point
 // in space, this routine returns the angle between the two :
 public static double angle( double[] vectorA, double[] vectorB ) {

double result = 0.0 ;
int index ;
// First work out the vector product of the two points :
for ( index = 0 ; index < vectorA.length ; index++ )
    result += ( vectorB[index] * vectorA[index] ) ;

// Then divide the product by the product of the two vector
// magnitudes, and return the inverse cosine of the result :
return Math.acos( result / ( v_mag( vectorA ) * v_mag( vectorB ) ) ) ;
 }

 // Generates the 2d representatin of this object in the
 // given set of dimensions (set true if a dimension is
 // included in the nd – 2d calculation)
 public xyPoint xy( ndPoint initial, ndPoint goal, double display_scale,
       double display_angle, int display_width ) {

xyPoint result = new xyPoint() ;
```

```
    double magnitude = magnitude( initial.position, position ) ;

    // Calculate the reference vector
    int counter ;
    double[] reference = new double[initial.position.length] ;
    for ( counter = 0 ; counter < reference.length ; counter++ )
        reference[counter] = goal.position[counter] - initial.position[counter] ;

    double[] current = new double[initial.position.length] ;
    for ( counter = 0 ; counter < reference.length ; counter++ )
        current[counter] = position[counter] - initial.position[counter]          ;

    double angle = angle( initial.position, position ) - display_angle ;

    // From the input data, calculate cartesian
    // coordiantes for the result data :
    result.x = (int)((magnitude * Math.cos( angle ))
            * display_scale) ;
    result.y = (int)((magnitude * Math.sin( angle ))
            * display_scale) ;

    return result ;
     }
}
```

## 12.7 The server program `SimServ.c`

```
/* Neural Network Simulator Server Program

    C Version Connecting directly to stubs

    This Server allows the output from the C based
    neural network simulator to be forwarded to
    a client applet, when that applet involkes the
    server.

    Duncan McPherson,   9/02/00  */

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/systeminfo.h>
#include <sys/types.h>
```

```
#include <time.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/un.h>
#include <unistd.h>
#include <netinet/in.h>
#include <netdb.h>

// Inclusion from network simulator
//#include "vector.h"

// External function used to initialise the simulator
extern void simulator_initialise() ;

#define SERVER "/tmp/serversocket"
#define MAXMSG 512
#define PORT   5555
#define ANNSIM "SUBGOALAPPLET" // Welcome sent by the applet
#define HELLO "ANNSIMSERVER" // Reply sent by the server
#define SIMSTART "SIMSTART" // Parameterized start request

// Reads commands from the applet that tries to connect to the port :
int read_from_applet (int filedes) {
  char buffer[MAXMSG];
  int nbytes;

  nbytes = read (filedes, buffer, MAXMSG);
  if (nbytes < 0) {
       // Read error creates the following message :
       printf("Error on reading from the client\n") ;
       exit (EXIT_FAILURE);
  }
  else if (nbytes == 0)
       // End-of-file
       return -1;
  else {
       // Data read
       fprintf (stderr, "Server: got: '%s'\n", buffer);
       printf( "Hello command is %s of length %i\n", HELLO, strlen(HELLO) ) ;

       if ( strncmp( HELLO, buffer, strlen(HELLO) ) == 0 )
      fork_simulator( filedes ) ;

       return 0;
```

```
  }

}

int main (void) {

  extern int make_socket (unsigned short int port);
  int sock;
  fd_set active_fd_set, read_fd_set;
  int i;
  struct sockaddr_in clientname;
  size_t size;

  // Create the socket and set it up to accept connections
  sock = make_socket( PORT );
  if (listen (sock, 1) < 0) {
      printf("Error on listen for socket\n");
      exit (EXIT_FAILURE);
  }

  // Initialize the set of active sockets.
  FD_ZERO( &active_fd_set );
  FD_SET( sock, &active_fd_set );

  while (1) {
      // Block until input arrives on one or more active sockets
      read_fd_set = active_fd_set;
      if (select (FD_SETSIZE, &read_fd_set, NULL, NULL, NULL) < 0) {
     printf("select\n");
     exit (EXIT_FAILURE);
      }

      // Service all the sockets with input pending
      for (i = 0; i < FD_SETSIZE; ++i)
     if (FD_ISSET (i, &read_fd_set)) {
         if (i == sock) {
        // Connection request on original socket
        int new;
        size = sizeof (clientname);
        new = accept (sock,
            (struct sockaddr *) &clientname,
            &size);
        if (new < 0) {
            printf ("accept\n");
```

```
                exit (EXIT_FAILURE);
        }
            fprintf (stderr,
                "Server: connect from host %s, port %hd.\n",
                inet_ntoa (clientname.sin_addr),
                ntohs (clientname.sin_port));
            FD_SET (new, &active_fd_set);
             } else {
            // Data arriving on an already-connected socket
            if (read_from_applet (i) < 0) {
                close (i);
                FD_CLR (i, &active_fd_set);
            }
             }
        }
    }
}

// Forks off an ANN simulator process with inputs given across the web
int fork_simulator( int filedes ) {

     int pid, nbytes ;

     //nbytes = read (filedes, buffer, MAXMSG);
     // If this process is the newly forked simulator process, act on that
     if((pid=fork())==0) {

    //dup2(www_chan_in[0],fileno(stdin));        Attach www_chan_in to stdin

    printf("Server thread starts here!!") ;
    write( filedes, "testing", 7 ) ;
    //execl("./" SERVER,SERVER,(char *)0);
    printf("The Server is exiting for this process - %i\n", pid ) ;

    dup2(filedes,fileno(stdout));          // Attach www_chan_out to stdout

    // Call the main function of the simulator to start the ball rolling
    simulator_initialise() ;

    exit(EXIT_FAILURE) ;
     }
     printf("PID - %i continues \n", pid) ;
}
```

```
// Create a socket to the desiered port on this machine
int make_socket (unsigned short int port) {
    int sock;
    struct sockaddr_in name;

    // Create the socket here :
    sock = socket (PF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
   printf("socket creation error\n");
   exit (EXIT_FAILURE);
    }

    // Give the socket a name
    name.sin_family = AF_INET;
    name.sin_port = htons (port);
    name.sin_addr.s_addr = htonl (INADDR_ANY);
    if (bind (sock, (struct sockaddr *) &name, sizeof (name)) < 0) {
   printf("Error on socket bind\n");
   exit (EXIT_FAILURE);
    }

    return sock;
}

// Initialises a socket that has allready been created
// to a specific network address.
void init_sockaddr (struct sockaddr_in *name,
            const char *hostname,
            unsigned short int port) {
    struct hostent *hostinfo;

    name->sin_family = AF_INET;
    name->sin_port = htons (port);
    hostinfo = gethostbyname (hostname);
    if (hostinfo == NULL) {
   fprintf (stderr, "Unknown host %s\n", hostname);
   exit (EXIT_FAILURE);
    }
    name->sin_addr = *(struct in_addr *) hostinfo->h_addr;
}
```

# Chapter 13

# Status Report

At the end of the implementation phase most software functions described in the initial plan along with the enhancements described in the revised timetable chapter have been completed. Both the JSAND and Subgoal applets are able to operate correctly in their client server configurations, displaying data in real time sourced from training programs provided by the Dr Weir's research group.

The network editor function of the JSAND application build has not yet been completed, however since JSAND may import textual descriptions of a neural network this function is not critical for the end user to be able to make full use of the JSAND software. Instead of editing weight values of neural networks within JSAND using slider bars as originally envisaged, users can edit textual description files of neural networks instead. JSAND does allow the number of hidden units in a neural network to be edited, weight values to be randomised and the bias level to be changed.

The cubic spline interpolation module of the subgoals display applet has also not been implemented, as under normal training circumstances enough output state data is generated to plot a smoothly curved path between initial and goal states.

One enhancement to the Java canvas display code of both the JSAND and Subgoals software was to enable the use of double buffered animation. Though this has allowed for flicker free rotation and plotting of new subgoals it has been found to prohibit vectored output of the

display using the appletviewer's postscript print function. Since the double buffer code plots the display first into an image buffer, the applet viewer generates postscript output using the entire bitmap image buffer. A work around for this would be to disable the sections of code allowing this enhancement.

Since both neural network training programs supplied by the University of St Andrews require configuration data to be input from the standard input stream at startup, the server program must be executed with the appropriate configuration data piped as input. This has the disadvantage that the server may only be used once, and then must be restarted before another applet may make use of that server's training program. This could be overcome by the training programs reading configuration data from a file allowing the server to operate unattended, forking off new training programs to service applet requests.

# Bibliography

[1] Horstmann C.S. and Cornell G. *Core Java - Volume 1 - Fundamentals*. Prentice Hall PTR & Sun Microsystems Press, 1997.

[2] Flanagan D. *Java In A Nutshell*. O'Reilly & Associates, Inc. (USA), 1997.

[3] Muller B. Reinhardt J. *Neural Networks, an introduction*. London : Springer-Verlag, 1990.

[4] Storer J. Island - interactive single-layer linear activation network display. Technical report, The School of Computer Science, University of St Andrews, The North Haugh, St Andrews, UK, 1994.

[5] Franklin S. *Artificial Minds*. London : MIT Press pp 121-140, 1995.

[6] Gorse T., Shepherd A. J., and Taylor J. G. The new era in supervised learning. *Neural Networks*, 10(2):345–352, 1995.

[7] Jackson T. and Beale R. *Neural Computing*. London : Instutute of Physics Publishing, 1990.