Better FSAs Through Clustering

Investigating clustering-while-training of neural networks using standard and novel methods.

MLRC Technical Report
12th October, 2001

Authors: Dylan Muir (dr.muir@qut.edu.au)[1]
Michael Towsey (Michael.Towsey@gmd.de)[2]

1 Machine Learning Research Centre
(Smart Devices Laboratory)
Queensland University of Technology
Level 4, 126 Margaret street
Brisbane, QLD, 4002
AUSTRALIA

2 Institut fur Autonome intelligente Systeme (AiS)
Fraunhofer Gesellschaft
Sankt Augustin
GERMANY

Abstract

This technical report outlines work done between July 1998 and February 2001 in the Machine Learning Research Centre at QUT.

Research was performed on rule extraction from a spoken-language corpus. Recurrent neural networks (RNNs) were trained using a novel architecture based on work by Das & Moser [1998]. This architecture performed on-line spherical clustering of hidden layer activations to encourage clean state induction. The cluster codebooks generated during training were used to extract a Finite State Automaton (FSA) representing the learned rules.

These FSAs were smaller, and had improved generalisation prediction rates compared to FSAs extracted from RNNs with no on-line clustering. It was noted that the clusters of activations were elongated, and might be better represented by ellipsoidal clusters. It was hoped that this would further reduce the number of FSA states while providing similar or improved prediction rates.

A second-order distance metric was developed based on work by Lipson & Siegelmann [1998]. A new clustering algorithm was developed to use this distance metric with some promising results, but was not perfected by the end of the research.

The source code for a suite of utilities developed during the course of the research is also included in this report.

Keywords

Clustering algorithms, distance metrics, correlation matrix, neural networks, natural language processing, spoken language, on-line clustering, rule extraction, FSAs, recurrent networks.

Table of Contents

Abstract iii
Keywords iii
Table of Contents v
Table of Figures vii
Introduction 1
Part 1 – Clustering Algorithms and Distance Metrics 3
1.0 Clustering Theory and Definition of Terms 3
1.1 Pattern Vector 3
1.2 Distance 3
1.3 Cluster 4
2.0 Distance Metrics 5
2.1 Euclidean Distance Metric 5
2.2 Second Order Distance Metric 7
2.2.1 Second Order Distance Implementation Issues 11
2.3 Identifying Badly Formed Clusters 12
3.0 Clustering Algorithms 15
3.1 Modified Adaptive Forgy's Algorithm 15
3.2 Adaptive Order-2 Clustering Algorithm 17
4.0 Data Sets 21
4.1 Benchmark Set 21
4.1.1 Data (Benchmark) 22
4.2 Discretionary Set 23
4.2.1 Data (Discretionary) 24
5.0 On-line Clustering While Learning 27
6.0 FSA Extraction and Generation 29
7.0 Results 31
7.1 Training 31
7.2 Outcomes 32
8.0 References 35
Part 2 – Software 37
9.0 Software 37
9.1 Software Table of Contents 37
9.2 Applications and Utilities 39
9.2.1 tlearn 39
9.2.2 tlearn's Internal Structure 42
9.2.3 dstat 56
9.2.4 SymStrip 64
9.2.5 MakeFSA 85
9.2.6 pattern 96

9.2.7 vector 100
9.2.8 tlbe 109
9.2.9 sclust 115
9.2.10 o2clust 126
9.2.11 tlavq 138
9.3 Source Code Modules 186
9.3.1 StdDefs 186
9.3.2 2darray 186
9.3.3 gauss 188
9.3.4 RunAvg 189
9.3.5 htable 191
9.3.6 smdarray 197
9.3.7 TokenLst 204
9.3.8 TokScan 207
9.3.9 vector_utils 209
9.3.10 vector_read 215
9.3.11 cluster 217
9.3.12 corr_matrix 222

Table of Figures

Figure 1-1 A modified set of clusters [Zupan 1982, p39] that are poorly represented by their centroids (indicated by circles). 4
Figure 2-1 Hyper-planes separating spherical clusters in a two dimensional space. 5
Figure 2-2 The Euclidean distance surface of a single spherical cluster at (8, 10). 6
Figure 2-3 Several spherical clusters. The decision planes can be seen by looking at the minima of these surfaces. 6
Figure 2-4 The decision planes are visible as the intersections of the surface contours. 6
Figure 2-5 The correlation distance surface of a spherical cluster at (50, 50). 9
Figure 2-6 The correlation distance surface of an elliptical cluster at (80, 80). 9
Figure 2-7 Several correlation clusters. 10
Figure 2-8 The parabolic decision surfaces can be seen as the intersection of the distance contours. 10
Figure 2-9 An elliptical cluster. 12
Figure 2-10 The distribution of correlation for a well-defined cluster. Standard deviations are shown. 12
Figure 2-11 A difficult cluster. 12
Figure 2-12 The distribution of correlation for a poorly defined cluster. Standard deviations are shown. 12
Figure 4-1 The benchmark data set [Zupan 1982, p57]. 21
Figure 4-2 A contrived strongly ellipsoidal data set. 23
Figure 5-1 The proposed online clustering while learning architecture. 27
Figure 6-1 The hidden layer activations for a recurrent network trained on a spoken language corpus. 29
Figure 7-1 The result of training a single network with the on-line clustering architecture compared to a network trained without on-line clustering. The thick dark lines are the prediction scores for the AVQ-hi network on (top to bottom) training and test data. The thinner lines are the prediction scores for the networks trained without clustering, for the same input data. 32
Figure 7-2 The result of FSA extraction from the three sets of networks. 33
Figure 9-1 Example network configuration for tlearn. 40
Figure 9-2 tlearn quick reference page. 41
Figure 9-3 Plot of classification a vs. RMS prediction error given by eqtn (8). 138

Introduction

This report investigates the use of two clustering methods on various data sets. To support this, the derivation and explanation of a few mathematical tools is necessary. Various issues to do with clustering are discussed, along with the description of two distance metrics. One of these (Euclidean distance) is conventional; the investigation of the other (second order distance) forms the major content of this report.

Das and Moser [1998] describe a method where clustering is performed while training a neural network, in order to expedite the learning of clean, accurate finite state automata (FSA). Our research adapts this method to data sets not created by an FSA. This research was also motivated by the observation that activations in the hidden unit space of a simple recurrent network (SRN) form elongated or ellipsoidal clusters (See Figure 6-1). It was hoped that using a clustering method able to extract ellipsoidal clusters would allow a more precise FSA to be extracted.

Lipson and Siegelmann [1998] describe the use of a correlation matrix to allow a neuron to map higher-order information, such as orientation and scaling. Our research develops the correlation matrix representation into a clustering tool that classifies neural-network data sets more accurately than traditional spherical methods. This clustering tool is then used in an on-line training structure similar to that described by Das and Moser [1998].

Two data sets are used to examine and compare the clustering algorithms. A benchmark set extracted from Zupan [1982] is used, as well as another contrived data set designed for the identification of elliptical clusters. Some results obtained by training on a spoken-language data set are discussed. Finally, the source code is given for the clustering algorithms as well as a suite of utility programs developed in the course of the research.

Part 1 – Clustering Algorithms and Distance Metrics

1.0 Clustering Theory and Definition of Terms

1.1 Pattern Vector

Each point (pattern) in the data set over which we wish to perform clustering can be represented as a vector $\bar{x}_i$ in n-dimensional space.

$$\bar{x}_i = \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,n} \end{bmatrix}$$

The dimensions of this space need not correspond to the traditional Cartesian dimensions. Each dimension may be any measurable property of the data set, as long as this representation is consistent across the entire set. For the purposes of visualisation and interpretation of results, it is still necessary to map the data into a 2D space for display.

It is worthwhile to note that some clustering methods are sensitive to the distribution of data within the set. For example, if a dimension in the data space represents a Boolean value (i.e. the dimension only ever takes the values 0 and 1), this may have a large impact on the success of a clustering method that assumes a continuous distribution over the dimension.

1.2 Distance

The distance metric chosen for use in clustering has a strong effect on the generation of clusters which is not explicitly obvious. In effect, the distance metric determines the perception of the clusters by the algorithm, which operates without the benefit of the sophisticated neural clustering software we use for vision. The effectiveness of most clustering algorithms varies considerably with different distance metrics.

If data is represented as points in a three-dimensional space, it is easy to visualise our intuitive definition of distance separating objects in the real world. This is Euclidean distance, calculated by

$$d_{eu}\left(\bar{x}_i, \bar{x}_j\right) = \left[\sum_{k=1}^{n} \left(x_{i,k} - x_{j,k}\right)^2\right]^{\frac{1}{2}}$$ **Equation 1**

where $\bar{x}_i$ and $\bar{x}_j$ are the vectors i and j between which the distance is calculated and k=1..n specifies each of n dimensions.

1.3 Cluster

A cluster is a number of patterns grouped together according to some rules. Zupan [1982, p4] states

> "the goal of clustering is to find groups containing objects [patterns] most homogeneous within the group, while at the same time groups are heterogeneous between themselves as much as possible."

Cluster representations can be stored in various forms depending on the underlying distance metric. Clusters generated with Euclidean distance are usually represented by the centroid of the cluster. In many situations, this provides an inadequate representation of the cluster. Figure 1-1 shows some clusters for which representation by a single point is particularly bad.

Higher order distance metrics may alleviate this problem by using a cluster representation that encapsulates the shape of the cluster.

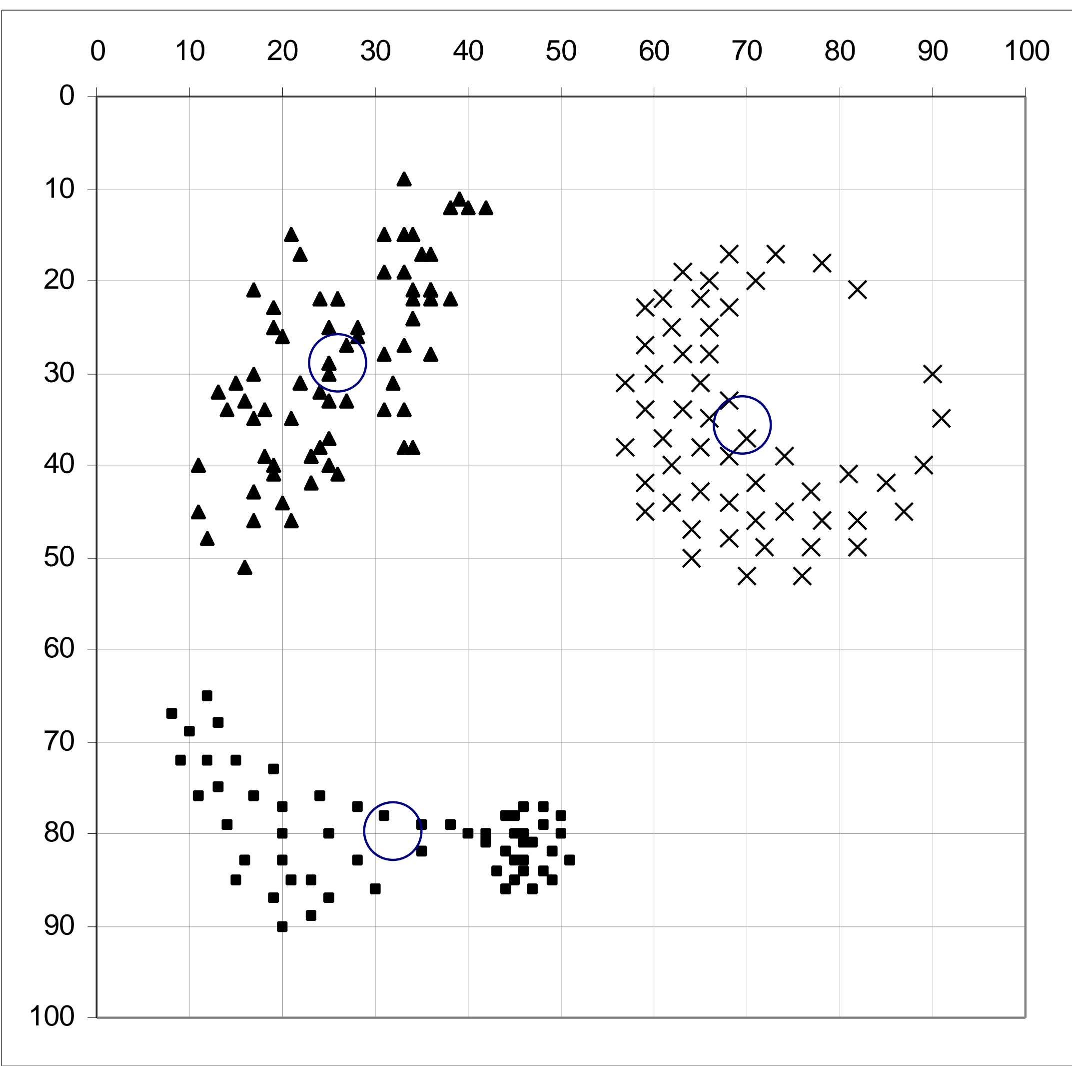


Figure 1-1 A modified set of clusters [Zupan 1982, p39] that are poorly represented by their centroids (indicated by circles).

2.0 Distance Metrics

The two metrics examined in the report are compared in terms of the shapes that the resulting clusters adopt, and in terms of the decision surfaces between adjacent clusters.

2.1 Euclidean Distance Metric

As stated previously, the Euclidean distance given by eqtn (1) corresponds to our real-world definition of the distance between two objects. Using a Euclidean metric will result in clusters that are hyper-spherical (in n dimensions). The decision surfaces are hyper-planes in the data space.

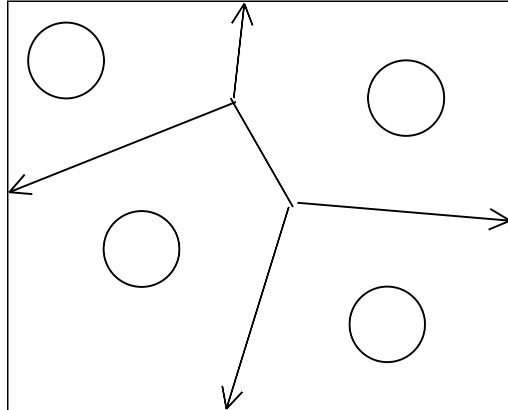


Figure 2-1 Hyper-planes separating spherical clusters in a two dimensional space.

Figure 2-1 shows flat decision surfaces resulting from a number of Euclidean cluster centroids in two dimensions.

Figure 2-2 shows that the equidistance surface of a single cluster forms an inverted cone with the vertex at the centroid. Horizontal cross-sections of this surface form a series of equally spaced concentric circles surrounding the centroid. Compare this with Figure 2-5, which shows a similar measurement taken with a second order distance metric.

Figure 2-3 shows the intersecting equidistance surfaces of several co-existing clusters.

Figure 2-4 shows the decision surfaces for the situation presented in Figure 2-3. The graph drawn represents the minima of the cones in Figure 2-3. The decision surfaces are the lines formed by the intersection of the cones.

Figure 2-2 The Euclidean distance surface of a single spherical cluster at (8, 10).

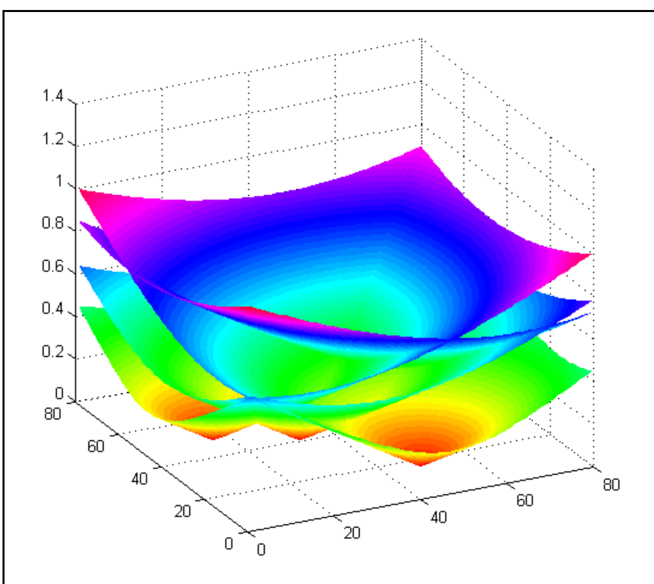


Figure 2-3 Several spherical clusters. The decision planes can be seen by looking at the minima of these surfaces.

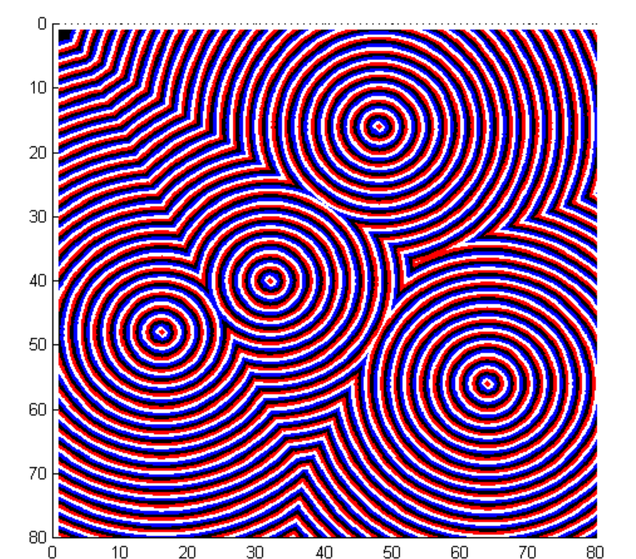


Figure 2-4 The decision planes are visible as the intersections of the surface contours.

2.2 Second Order Distance Metric

The second order distance measure is not actually taken between two points in data space, but is taken between a point and a cluster. Since this is the most common use of a distance metric in cluster analysis, this distinction does not pose too many problems.

The order-2 measure indicates how well a test point "fits" the data in an existing cluster. The training vectors are converted to homogenous form by appending a 1 to each vector:

$$\overline{X}_i = \begin{bmatrix} \overline{x}_i \\ 1 \end{bmatrix} \qquad \textbf{Equation 2}$$

This dispenses with the need to normalise each cluster by moving its centroid to the origin.

A correlation matrix is generated for each cluster, by summing the covariance matrix of each point within the cluster and normalising for cluster size.

$$R_{h,j} = \sum_{i=1}^{size(j)} \overline{X}_i \overline{X}_i^T \qquad \textbf{Equation 3}$$

where $R_{h,j}$ is the homogenous correlation matrix for cluster *j*, $\overline{X}_i$ is a vector belonging in cluster *j* and *size(j)* is the number of vectors currently belonging to cluster *j*. Note that $\overline{X}_i \overline{X}_i^T$ forms the cross product of the vector $\overline{X}_i$ with itself.

The second order distance d_{o2} is then measured by choosing a test point $\overline{x}$, and calculating

$$d_{o2}\left(\overline{x}, C_j\right) = \overline{X}^T R_{h,j}^{-1} \overline{X} \qquad \textbf{Equation 4}$$

where C_j is the cluster to measure second order distance to and $R_{h,j}$ is the homogenous correlation matrix generated for C_j. This gives a measurement similar to Euclidean distance, in that a low value for d_{o2} denotes a high similarity or closeness between the test point and the cluster.

The correlation matrix representing clusters generated using eqtn (3) store information about the extent of the cluster along each dimension in the data space. This representation is able to describe elongated clusters, whereas using the Euclidean distance d_{o1}, these clusters would need to be represented by multiple Euclidean centroids.

Figure 2-5 shows a perfectly spherical cluster as represented by a correlation matrix. The second order equidistance surface is shown. The surface is bowl shaped; the sides of the surface are parabolic. Compare this graph with Figure 2-2, where the surface is conical. Horizontal slices (contours) through this surface will give a series of concentric circles, similar to the Euclidean

equidistance surface in Figure 2-2. However, the spacing of the contours will not be even; the contours become more closely spaced further away from the center of the cluster.

Figure 2-6 shows an elliptical cluster as represented by a correlation matrix. The equidistance surface is once again bowl shaped, but is stretched along the primary axis of the cluster. The Euclidean equidistance surface of this cluster would be a cone similar to Figure 2-2. Horizontal slices through this surface will give a series of concentric ellipses. This is the natural shape of clusters represented in the method described here.

Figure 2-7 shows the second order equidistance surfaces of several co-existing clusters.

Figure 2-8 shows the decision surfaces for the situation presented in Figure 2-7. The graph is the second order equidistance contours of the minima of the surfaces. The decision surfaces are hyper-parabolas, evident at the intersection of the contours.

Figure 2-5 The correlation distance surface of a spherical cluster at (50, 50).

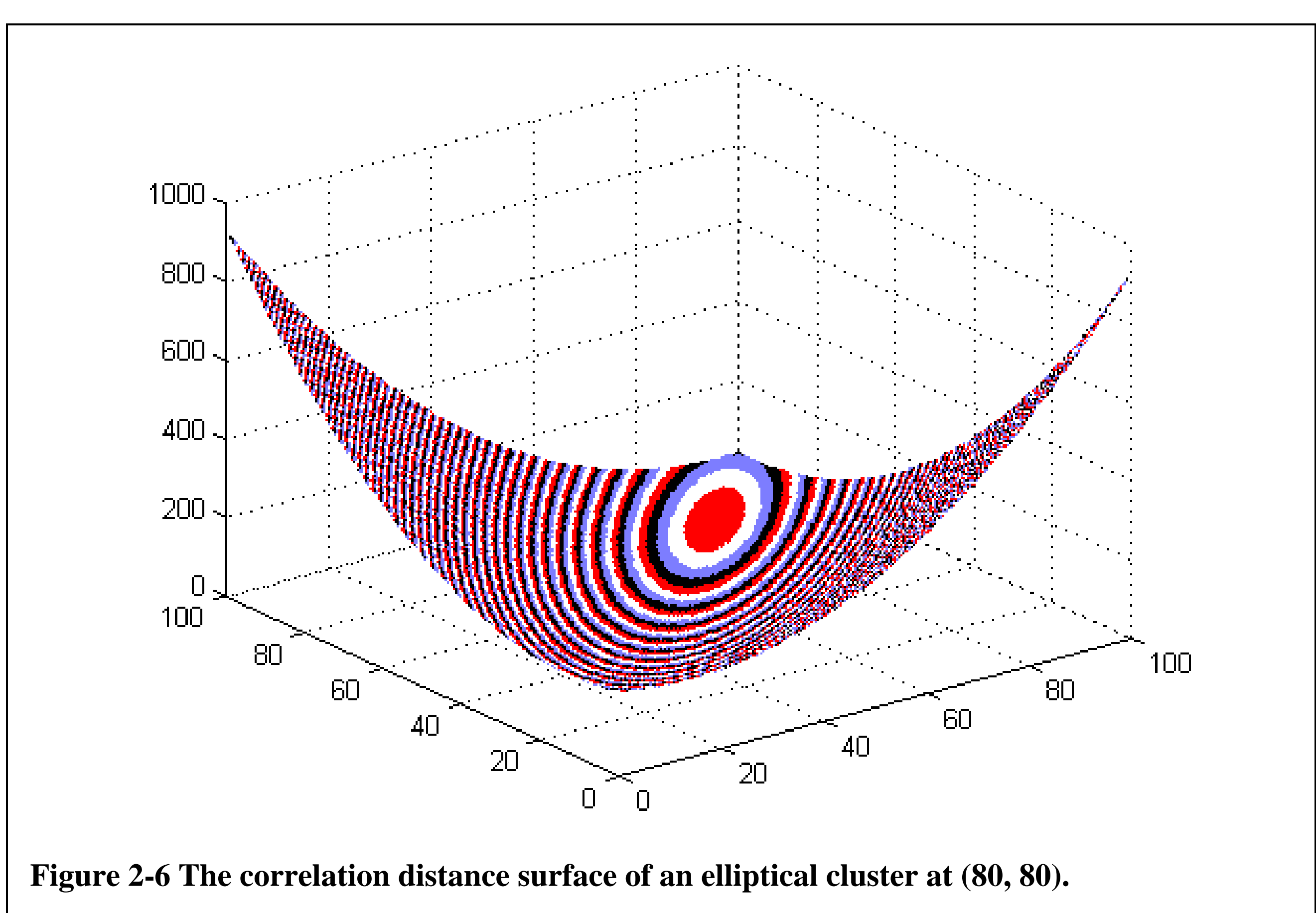


Figure 2-6 The correlation distance surface of an elliptical cluster at (80, 80).

Figure 2-7 Several correlation clusters.

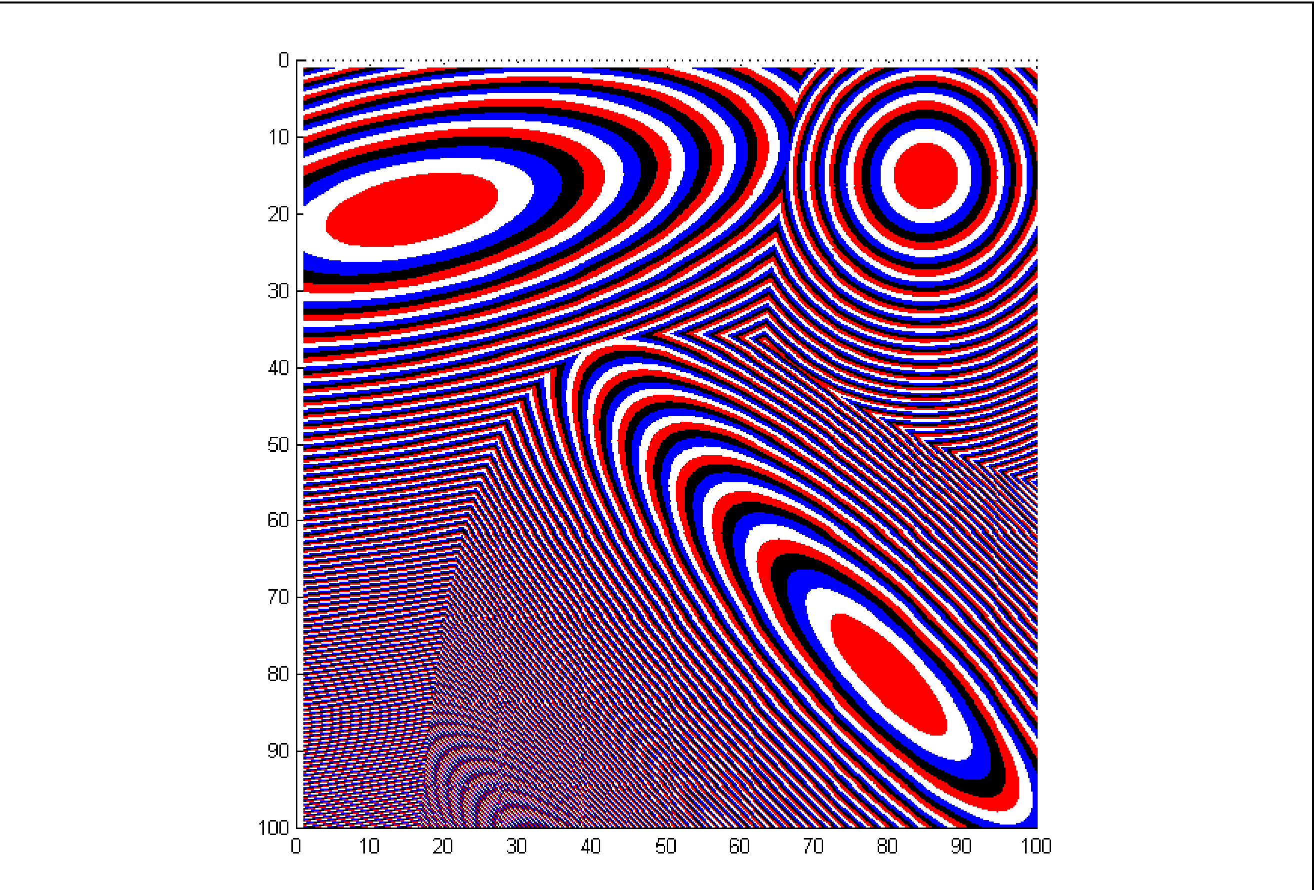


Figure 2-8 The parabolic decision surfaces can be seen as the intersection of the distance contours.

2.2.1 Second Order Distance Implementation Issues

The primary distinction between Euclidean measurement and second order measurement is that the second order measure cannot be taken between two points. The obvious solution to this is to generate a cluster consisting of only one point. Unfortunately, due to the method of generating the correlation matrix, clusters with a single point always result in singular correlation matrices. This precludes using the second order distance metric, as the measurement requires the correlation matrix to be inverted. Clusters with fewer than four points will usually generate matrices that are either singular or very badly scaled.

One attempted solution was to generate a very small (in area) hyper-spherical cluster centered at the centroid of the small cluster. "Dummy" points can be generated at a small distance from the centroid along each dimension. The resulting clusters do not generate singular matrices, and approximate the second order distance measure one would expect from a single point. When the real cluster has accumulated more than a threshold number of points, the "dummy" points can be discarded. For the purposes of implementation, this threshold was taken at four points.

Due to the fundamental operation of the second order distance metric, clusters with a small area will usually give a greater d_{o2} (and thereby correlate worse) than a cluster with a large area. This causes test points to tend towards clusters that are comparatively far away (but large in area) rather than nearby, smaller clusters. This problem is alleviated by normalising the d_{o2} measure.

$$\hat{d}_{o2}\left(\bar{x}_i, C_j\right) = \frac{d_{o2}\left(\bar{x}_i, C_j\right)}{d_{av,j}} \qquad \textbf{Equation 5}$$

The second order distance is normalised by dividing the measure taken in eqtn (4) by the average distance from all points within the cluster to the cluster itself ($d_{av,j}$).

$d_{av,j}$ is generated using

$$d_{av,j} = \frac{\sum_{i=1}^{size(j)} d_{o2}\left(\bar{x}_{j,i}, C_j\right)}{size(j)} \qquad \textbf{Equation 6}$$

where $\bar{x}_{j,i}$ is a vector already belonging in C_j.

2.3 Identifying Badly Formed Clusters

When analysing the results of a clustering run on a two-dimensional data set, it is relatively easy to see whether the results are good or bad. Data sets with higher dimensionality are difficult to visualise in this way. To aid in developing a clustering method for use with the correlation distance measure, it would be useful to be able to identify bad clusters automatically. The clustering algorithm will then be able to reclassify points in a "bad" cluster to improve its shape.

The only information an algorithm has about a cluster is the distribution of points within it. A perfectly identified cluster is one where it is possible to draw a contour on the equidistance surface that closely matches the apparent boundary of the cluster, as a human would perceive it.

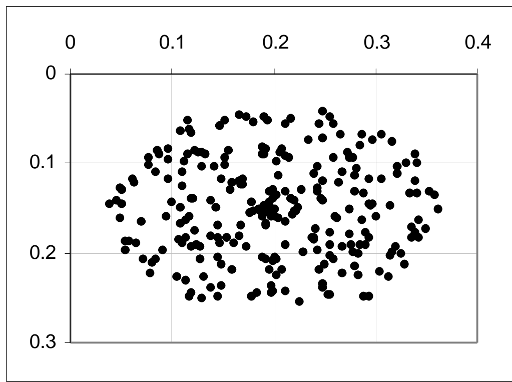


Figure 2-9 An elliptical cluster.

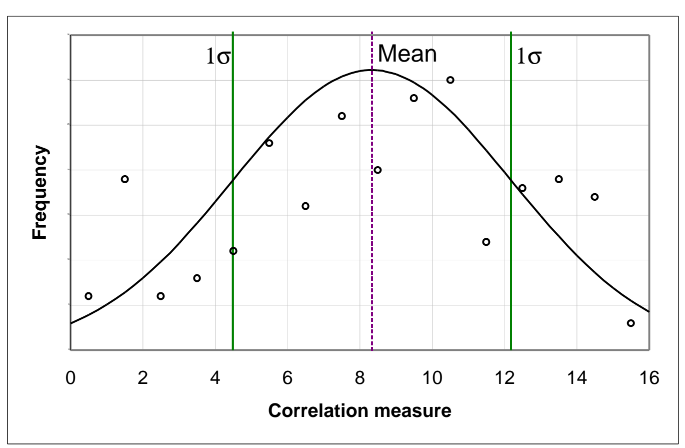


Figure 2-10 The distribution of correlation for a well-defined cluster. Standard deviations are shown.

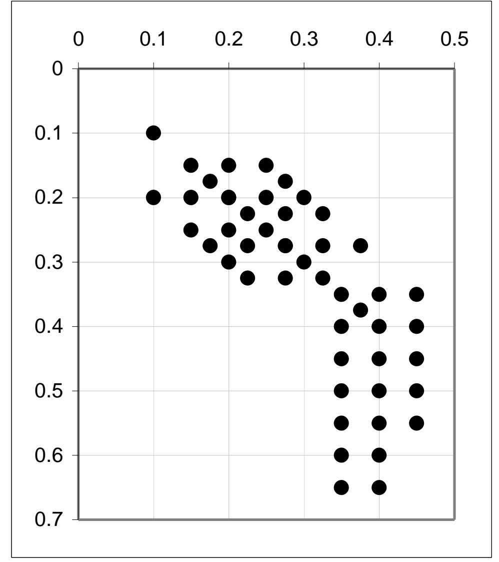


Figure 2-11 A difficult cluster.

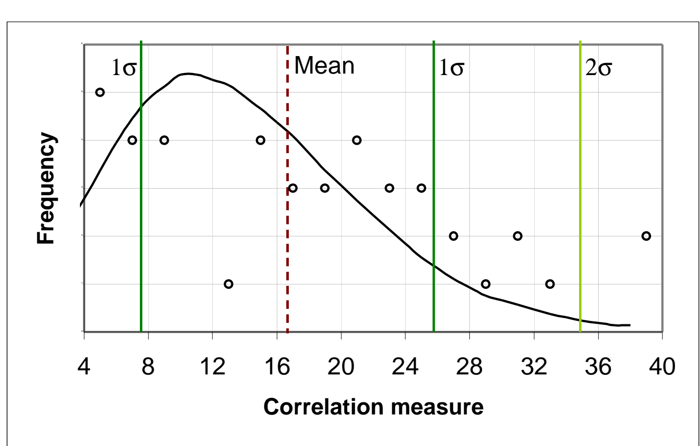


Figure 2-12 The distribution of correlation for a poorly defined cluster. Standard deviations are shown.

Figure 2-9 shows an elliptical cluster that is able to be well represented by the correlation matrix. The distribution of second order distance measured using eqtn (4) within this cluster is shown in Figure 2-10. All the points lie within two standard deviations of the mean.

Figure 2-11 shows a cluster that cannot be accurately represented by a second order correlation matrix. The distribution of second order distance within the poorly defined cluster is skewed to the right, as shown in Figure 2-12. This means that relatively more points have a bad correlation within the cluster, compared to the elliptical cluster. We can test how well a test point fits into a cluster by picking a threshold number of standard deviations above the mean distance as a criterion.

3.0 Clustering Algorithms

3.1 Modified Adaptive Forgy's Algorithm

For Order-1 (Spherical) Vector Quantisation

This algorithm is modified from a description of Forgy's algorithm in Das & Moser [1998]. It is an adaptive & deterministic algorithm. Clusters are represented by the arithmetic mean (centroid) of the vectors they contain. The algorithm uses Euclidean distance for all distance measures. Modifications were made to the new cluster criterion and the termination criterion, at lines 1 and 2.

- Define $\bar{x}_i$: current training vector
- Define C_{near} : cluster with centroid closest to $\bar{x}_i$
- Define d_{av} : average distance from all vectors to their classified cluster centroids
- Define d_{near} : distance from $\bar{x}_i$ to C_{near}
- Define e_{prev} : previous classification error
- Define e_{curr} : current classification error
- Define p_{dist} : a parameter controlling the creation of new clusters (typical value 0.5)
- Define p_{stop} : a parameter controlling when to stop clustering (a small positive number, typical value 0.0001)

- Add all training patterns to a single cluster
- Take the vector furthest from the centroid and make a new cluster
- Calculate e_{curr}

- **repeat**
  - $e_{prev} = e_{curr}$
  - **for** $\bar{x}_i$:= each vector in the training set
    - Calculate d_{av}
    - Find C_{near} and calculate d_{near}
    - (1) **if** $(d_{near} > p_{dist} \cdot d_{av})$ **then**
      - make a new cluster using $\bar{x}_i$
    - **else**
      - move vector into C_{near}
    - **end if**
  - **end for**
  - Calculate e_{curr}
- (2) **until** $((e_{curr} < e_{prev}) \wedge ((e_{prev} - e_{curr}) < p_{stop}))$

3.2 Adaptive Order-2 Clustering Algorithm

Using an order-2 (Ellipsoidal) distance metric

This algorithm uses the second order distance metric for all distance calculations. Terms such as 'closest' and 'furthest' are referring to second order distance. Section 2.2 outlines this distance metric and cluster representation. The algorithm is both adaptive and deterministic.

Normalised second order distances ($\hat{d}_{o2}$) are used when comparing distances between clusters. For example, when finding the nearest cluster (C_{near}) to a test point, the comparisons are made between normalised second order distances.

The method for identifying badly classified vectors for reclassification described in section 2.3 is utilised in this algorithm. A test is made at line 1 to determine whether the vector $\bar{x}_i$ will change classification. If $\bar{x}_i$ will retain its current classification, a test at line 2 determines whether $\bar{x}_i$ is well classified in $C_{\bar{x}_i}$. If the second order distance from $\bar{x}_i$ to $C_{\bar{x}_i}$ is further than a threshold number of standard deviations above the mean for $C_{\bar{x}_i}$, it is badly classified in $C_{\bar{x}_i}$, and may be reclassified. A second-best cluster is identified (C_{2nd}), which is the cluster closest to $\bar{x}_i$ after $C_{\bar{x}_i}$. $\bar{x}_i$ is tested in a similar fashion against C_{2nd} at line 3. If this test succeeds, then $\bar{x}_i$ is sufficiently well classified in C_{2nd} to warrant placing it there. Otherwise, the creation of a new cluster using $\bar{x}_i$ is justified.

- Define $\bar{x}_i$: current training vector
- Define C_{near} : cluster which $\bar{x}_i$ is closest to using second order distance
- Define C_{2nd} : cluster which $\bar{x}_i$ is closest to, disregarding C_{near}
- Define $C_{\bar{x}_i}$: current classification of $\bar{x}_i$
- Define $\hat{d}_{av,\forall}$: average normalised second order distance from $\bar{x}_i$ to all existing clusters
- Define $d_{av,j}$: average second order distance from all vectors within C_j to C_j
- Define $d_{stddev,j}$: standard deviation of distance from all vectors within C_j to C_j
- Define d_{near} : second order distance from $\bar{x}_i$ to C_{near}
- Define d_{2nd} : second order distance from $\bar{x}_i$ to C_{2nd}
- Define e_{prev} : previous classification error
- Define e_{curr} : current classification error
- Define p_{stddev} : a parameter determining the number of standard deviations from $d_{av,j}$ a vector $\bar{x}_{j,i}$ must be before it is considered badly classified (typical value 2.0)
- Define $s_{av,\forall}$: the average number of vectors per cluster
- Define s_j : the number of vectors in C_j
- Define p_{size} : a parameter determining what proportion of $s_{av,\forall}$ a cluster must have below which it is considered small (typical value 1.0)
- Define p_{new} : a parameter determining whether to create a new cluster containing an unclassified vector (a positive number less than 1.0)
- Define p_{stop} : a parameter controlling when to stop clustering (a small positive number, typical value 0.01)

- Make an initial cluster containing the single vector closest to the centroid of the data set
- **for** $\bar{x}_i$:= each vector in the training set
  - **If** $\left(\hat{d}_{near} >= p_{new} \cdot \hat{d}_{av,\forall}\right)$ **then**
    - make a new cluster using x_i
  - **else** *[$\bar{x}_i$ is classified sufficiently well in C_{near}]*
    - move $\bar{x}_i$ into C_{near}
  - **end if**
- **end for**
- Calculate e_{curr}

- **repeat**
  - $e_{prev} = e_{curr}$
  - **for** $\bar{x}_i$:= each vector in the training set
    - Find C_{near}
    - (1) **if** $\left(C_{near} \neq C_{\bar{x}_i}\right)$ **then**
      - Move $\bar{x}_i$ into C_{near}
    - **else** *[$\left(C_{near} = C_{\bar{x}_i}\right)$]*
      - Calculate $d_{\bar{x}_i}, d_{av,\bar{x}_i}$ and $d_{stddev,\bar{x}_i}$
      - (2) **If** $\left(\left(d_{\bar{x}_i} - d_{av,\bar{x}_i}\right) \leq \left(d_{stddev,\bar{x}_i} \cdot p_{stddev,\bar{x}_i}\right)\right)$ **then**
        - Leave $\bar{x}_i$ in $C_{\bar{x}_i}$
      - **else** *[$\bar{x}_i$ is badly classified in $C_{\bar{x}_i}$]*
        - Find C_{2nd} and calculate d_{2nd}
        - Calculate $d_{av,2nd}$ and $d_{stddev,2nd}$
        - (3) **If** $\left(\left(d_{2nd} - d_{av,2nd}\right) \leq \left(d_{stddev,2nd} \cdot p_{stddev,2nd}\right)\right)$ **then**
          - Move $\bar{x}_i$ into C_{2nd}
        - **else** *[$\bar{x}_i$ is badly classified in C_{2nd}]*
          - Make a new cluster using $\bar{x}_i$
        - **end if**
      - **end if**
    - **end if**
  - **end for**
  - Calculate e_{curr}
- **until** $\left(\left(e_{curr} < e_{prev}\right) \wedge \left(e_{prev} - e_{curr}\right) < p_{stop}\right)$

[optional procedure to eliminate small clusters]

- Calculate $s_{av,\forall}$
- **for** C_j := each cluster
  - Calculate s_j
  - **If** $\left(s_j < s_{av,\forall} \cdot p_{size}\right)$ **then**
    - cull C_j by reclassifying vectors
  - **end if**
- **end for**

4.0 Data Sets

4.1 Benchmark Set

This data set is used as a benchmark to compare the clustering methods. It has been transcribed from Zupan [1982, p57].

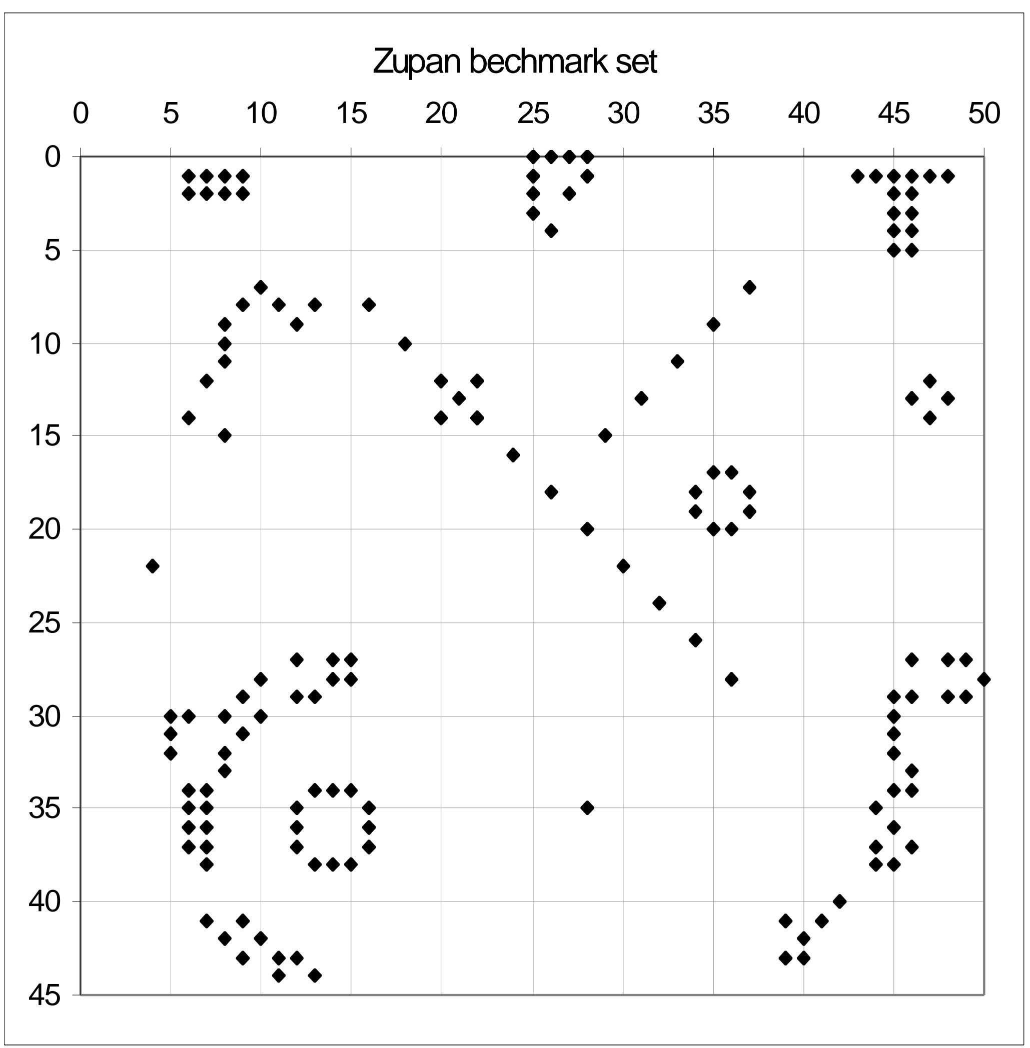


Figure 4-1 The benchmark data set [Zupan 1982, p57].

4.1.1 Data (Benchmark)

Benchmark data set from p57 of
Zupan, J. 1982, *Clustering of Large Data Sets*, John Wiley & Sons Ltd., Chichester.
Transcribed by Dylan Muir (dr.muir@student.qut.edu.au)
Date: 6th October, 2000

Point	x	y	Point	x	y	Point	x	y
1	6	1	51	24	16	101	7	37
2	7	1	52	26	18	102	7	38
3	8	1	53	28	20	103	13	34
4	9	1	54	30	22	104	14	34
5	6	2	55	32	24	105	15	34
6	7	2	56	34	26	106	16	35
7	8	2	57	36	28	107	16	36
8	9	2	58	29	15	108	16	37
9	25	0	59	31	13	109	12	35
10	26	0	60	33	11	110	12	36
11	27	0	61	35	9	111	12	37
12	28	0	62	37	7	112	13	38
13	25	1	63	47	12	113	14	38
14	25	2	64	46	13	114	15	38
15	25	3	65	48	13	115	7	41
16	26	4	66	47	14	116	8	42
17	28	1	67	35	17	117	9	41
18	27	2	68	36	17	118	10	42
19	43	1	69	34	18	119	9	43
20	44	1	70	34	19	120	11	43
21	45	1	71	37	18	121	12	43
22	46	1	72	37	19	122	11	44
23	47	1	73	35	20	123	13	44
24	48	1	74	36	20	124	28	35
25	45	2	75	4	22	125	39	41
26	45	3	76	15	27	126	39	43
27	45	4	77	15	28	127	40	43
28	45	5	78	14	27	128	40	42
29	46	2	79	14	28	129	41	41
30	46	3	80	13	29	130	42	40
31	46	4	81	12	29	131	44	38
32	46	5	82	12	27	132	44	37
33	10	7	83	10	28	133	45	38
34	9	8	84	9	29	134	46	37
35	11	8	85	10	30	135	45	36
36	8	9	86	9	31	136	44	35
37	12	9	87	8	30	137	45	34
38	8	10	88	6	30	138	46	34
39	8	11	89	5	30	139	46	33
40	7	12	90	5	31	140	45	32
41	6	14	91	5	32	141	45	31
42	8	15	92	8	32	142	45	30
43	13	8	93	8	33	143	45	29
44	16	8	94	6	34	144	46	29
45	18	10	95	6	35	145	46	27
46	20	12	96	6	36	146	48	27
47	22	12	97	6	37	147	49	27
48	21	13	98	7	34	148	50	28
49	20	14	99	7	35	149	49	29
50	22	14	100	7	36	150	48	29

4.2 Discretionary Set

The author has created this contrived data set with suggestions from Michael Towsey, to provide a set that is both strongly ellipsoidal and deliberately difficult to cluster.

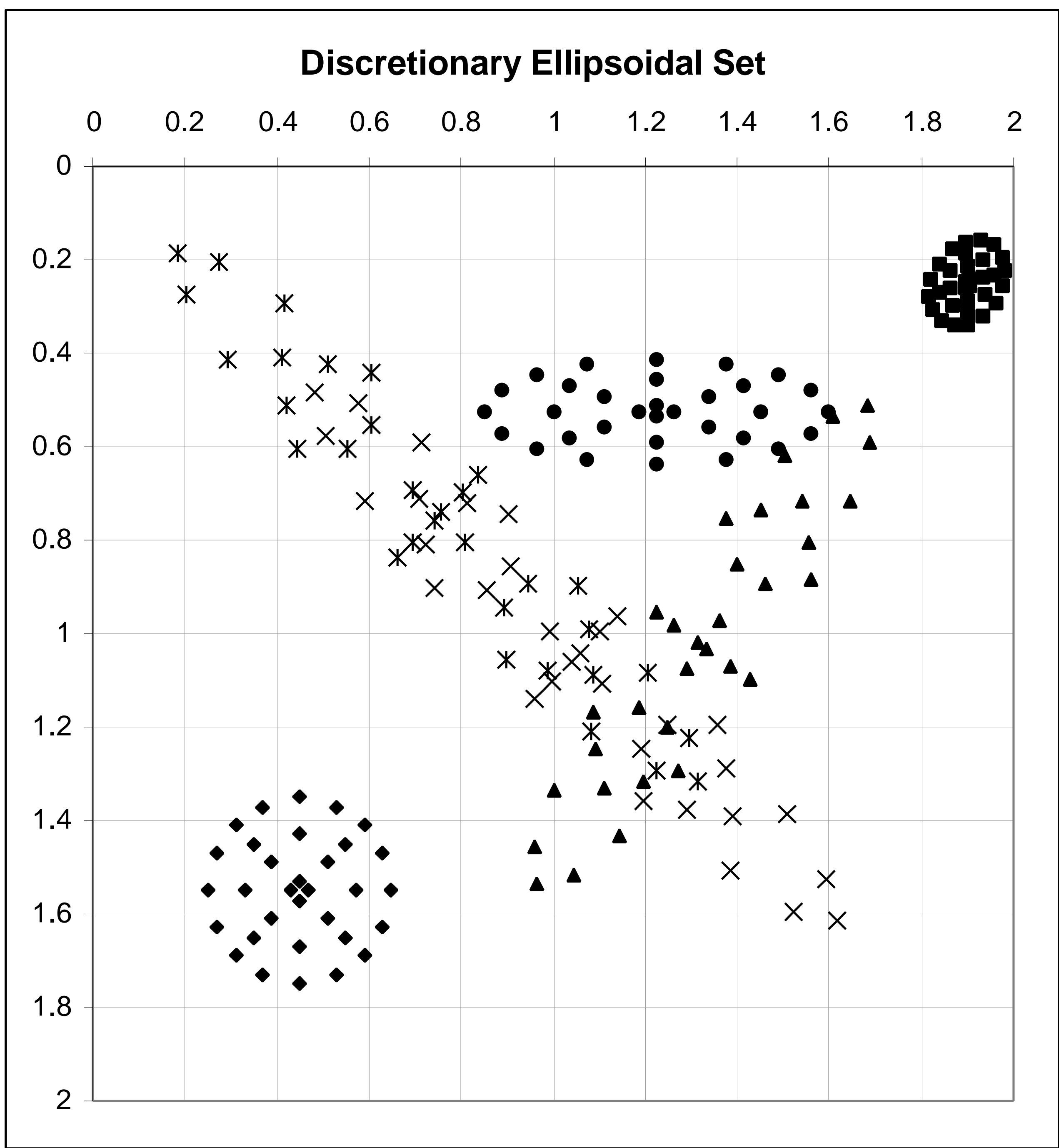


Figure 4-2 A contrived strongly ellipsoidal data set.

4.2.1 Data (Discretionary)

Discretionary ellipsoidal data set
Created by Michael Towsey (Michael.Towsey@gmd.de)
and Dylan Muir (dr.muir@student.qut.edu.au)
Transcribed by Dylan Muir (dr.muir@student.qut.edu.au)
Date: 8th October, 2000

Point	x	y	Point	x	y	Point	x	y
1	0.45	1.57	33	1.91	0.25	65	1.34	1.03
2	0.25	1.55	34	1.84	0.33	66	0.97	1.54
3	0.45	1.75	35	1.96	0.29	67	1.43	1.10
4	0.65	1.55	36	1.96	0.17	68	1.68	0.51
5	0.45	1.35	37	1.84	0.21	69	1.22	0.95
6	0.31	1.69	38	1.90	0.34	70	1.15	1.43
7	0.31	1.41	39	1.82	0.28	71	1.00	1.33
8	0.59	1.69	40	1.98	0.22	72	1.65	0.72
9	0.59	1.41	41	1.90	0.16	73	1.50	0.62
10	0.35	1.65	42	1.90	0.31	74	1.20	1.32
11	0.35	1.45	43	1.84	0.27	75	1.09	1.25
12	0.55	1.65	44	1.96	0.23	76	1.56	0.80
13	0.55	1.45	45	1.90	0.19	77	1.45	0.73
14	0.51	1.61	46	1.94	0.24	78	1.46	0.89
15	0.39	1.61	47	1.90	0.29	79	1.25	1.20
16	0.39	1.49	48	1.86	0.26	80	1.19	1.16
17	0.51	1.49	49	1.90	0.21	81	1.40	0.85
18	0.43	1.55	50	1.89	0.26	82	1.29	1.08
19	0.45	1.53	51	1.89	0.25	83	1.31	1.02
20	0.47	1.55	52	1.91	0.24	84	1.36	0.97
21	0.45	1.67	53	1.94	0.28	85	1.39	1.07
22	0.45	1.43	54	1.86	0.22	86	1.26	0.98
23	0.33	1.55	55	1.87	0.30	87	1.11	1.33
24	0.57	1.55	56	1.93	0.20	88	1.54	0.72
25	0.37	1.73	57	1.93	0.32	89	1.27	1.29
26	0.27	1.63	58	1.87	0.34	90	1.04	1.51
27	0.37	1.37	59	1.82	0.24	91	1.09	1.17
28	0.27	1.47	60	1.82	0.31	92	0.96	1.46
29	0.63	1.63	61	1.98	0.19	93	1.69	0.59
30	0.63	1.47	62	1.93	0.16	94	1.61	0.54
31	0.53	1.73	63	1.98	0.26	95	1.56	0.88
32	0.53	1.37	64	1.87	0.18	96	1.38	0.76

Point	x	y
97	1.04	1.06
98	0.48	0.48
99	0.96	1.14
100	1.62	1.62
101	1.14	0.96
102	0.59	0.72
103	0.72	0.59
104	1.38	1.51
105	1.51	1.38
106	0.72	0.81
107	0.81	0.72
108	1.29	1.38
109	1.38	1.29
110	1.19	1.25
111	0.85	0.91
112	0.91	0.85
113	1.25	1.19
114	0.99	0.99
115	1.06	1.04
116	1.11	1.11
117	1.00	1.10
118	1.10	1.00
119	0.71	0.71
120	1.39	1.39
121	0.74	0.90
122	0.51	0.58
123	0.90	0.74
124	0.58	0.51
125	1.52	1.59
126	1.59	1.52
127	1.20	1.36
128	1.36	1.20

Point	x	y
129	0.74	0.76
130	0.18	0.18
131	0.66	0.84
132	1.32	1.32
133	0.84	0.66
134	0.29	0.42
135	0.42	0.29
136	1.08	1.21
137	1.21	1.08
138	0.42	0.51
139	0.51	0.42
140	0.99	1.08
141	1.08	0.99
142	0.89	0.95
143	0.55	0.61
144	0.61	0.55
145	0.95	0.89
146	0.69	0.69
147	0.76	0.74
148	0.81	0.81
149	0.70	0.80
150	0.80	0.70
151	0.41	0.41
152	1.09	1.09
153	0.44	0.60
154	0.21	0.28
155	0.60	0.44
156	0.28	0.21
157	1.22	1.29
158	1.29	1.22
159	0.90	1.06
160	1.06	0.90

Point	x	y
161	1.23	0.54
162	0.85	0.53
163	1.23	0.64
164	1.60	0.53
165	1.23	0.41
166	0.96	0.60
167	0.96	0.45
168	1.49	0.60
169	1.49	0.45
170	1.04	0.58
171	1.04	0.47
172	1.41	0.58
173	1.41	0.47
174	1.34	0.56
175	1.11	0.56
176	1.11	0.49
177	1.34	0.49
178	1.19	0.53
179	1.23	0.51
180	1.26	0.53
181	1.23	0.59
182	1.23	0.46
183	1.00	0.53
184	1.45	0.53
185	1.08	0.63
186	0.89	0.57
187	1.08	0.42
188	0.89	0.48
189	1.56	0.57
190	1.56	0.48
191	1.38	0.63
192	1.38	0.42

5.0 On-line Clustering While Learning

Das and Moser [1998] outline a recurrent network architecture that incorporates online cluster analysis and soft classification while training. The architecture presented here is a hybrid of these ideas and the Elman recurrent architecture. Das and Moser use second-order connections in their architecture; the Elman architecture requires only first-order connections.

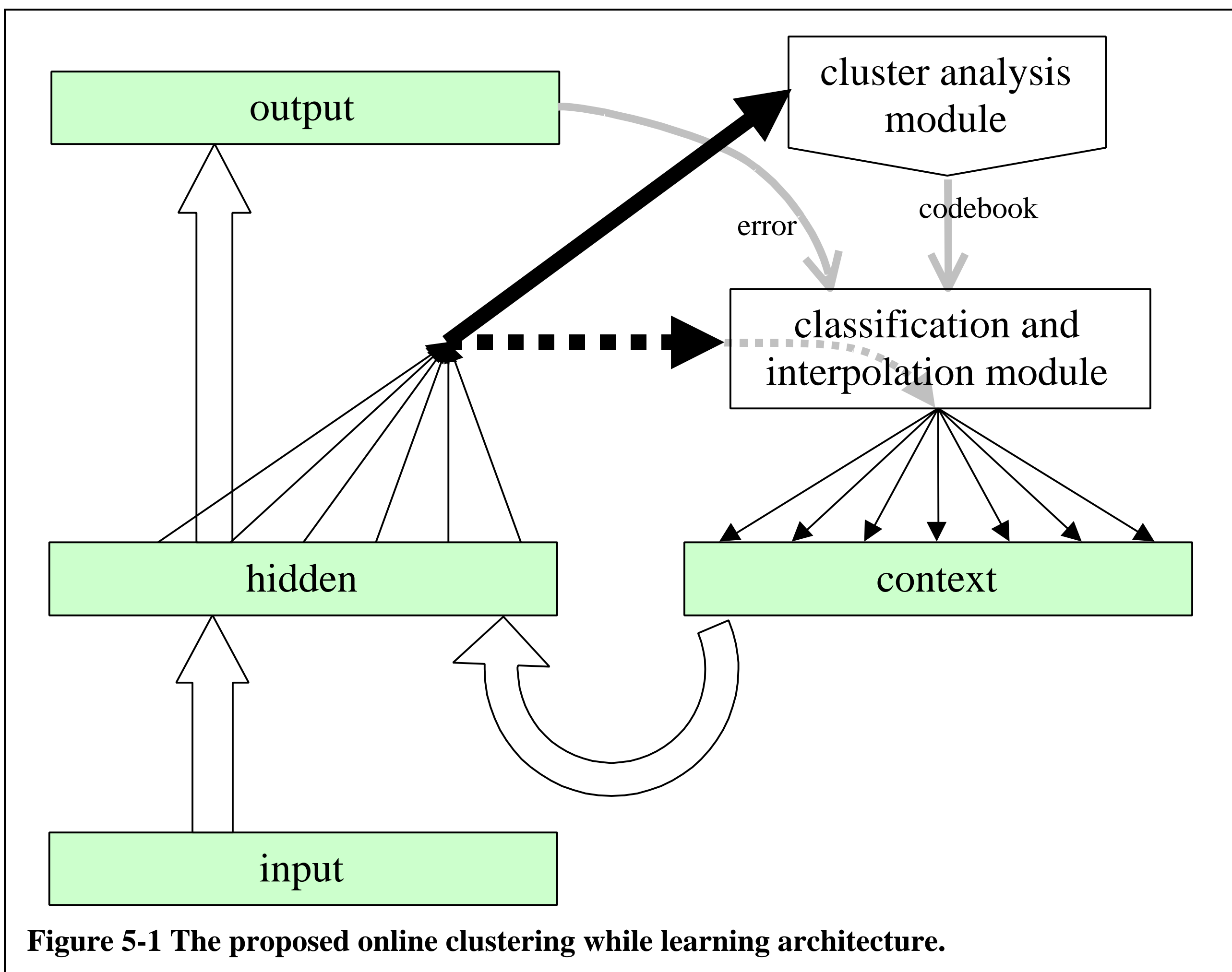


Figure 5-1 The proposed online clustering while learning architecture.

This clustering architecture relies on the assumption that when a recurrent net is trained on a language task, the states of a Finite State Automaton can be inferred by identifying the clusters of hidden unit activations.

In the standard Elman architecture, the hidden unit activations are copied 1-to-1 into the context layer. In the proposed online clustering architecture shown in Figure 5-1, the cluster analysis module identifies the clusters of hidden unit activations and generates a codebook. A classification and interpolation module sits between the hidden layer and the context layer. The hidden unit activations are interpolated with the classified activations before being copied into the context layer.

To facilitate learning, the system uses a ‘soft’ classification, as described by Das & Moser [1998]. The current prediction error of the network is used to generate a value a. This value determines the ‘strength’ of the classification. For a less than 1.0, the context unit activations will be an interpolation between the current hidden unit activations and the identified cluster that this set of activations belongs to.

$$\bar{c} = (1 - a)\bar{h} + a\bar{i}_{\bar{h}} \qquad \textbf{Equation 7}$$

where:

- $\bar{c}$ is the vector formed by the context layer activations
- $\bar{h}$ is the vector formed by the hidden layer activations
- $\bar{i}_{\bar{h}}$ is the centroid of the cluster that $\bar{h}$ classifies to
- a is some function of the current prediction error.

Any suitable function can be used to generate a. We use a variant of Das’ and Moser’s [1998] function

$$a = e^{(-f \cdot e_{corrected})} \qquad \textbf{Equation 8}$$

where:

- f is a parameter used to adjust the rate at which the ‘strength’ of clustering increases
- $e_{corrected}$ is the corrected mean prediction error, given by

$$e_{corrected} = \sqrt{\frac{mean\ squared\ output\ error}{num\ outputs}} \qquad \textbf{Equation 9}$$

Since the context layer does not sit between the hidden layer and the output layer, the error does not have to be propagated through it for learning.

6.0 FSA Extraction and Generation

Elman Simple Recurrent Networks (SRNs) have been shown to be able to infer finite state automata (FSAs) when trained on time-dependent data [Giles & Omlin, 1993a & 1993b]. Research has also been performed [Schellhammer et al., 1998] to investigate the extraction of previously unknown FSAs from networks trained on a simple language corpus.

This research relies on the assumption that a recurrent network trained on a language corpus will infer an FSA in the activations of the hidden layer. The states of the FSA are defined by performing cluster analysis on the set of hidden layer activations over the corpus. The resulting codebook can be used to determine what state the network is in at any time.

Figure 6-1 The hidden layer activations for a recurrent network trained on a spoken language corpus.

In Figure 6-1, the network has generated internal states that can be seen using principal components analysis to map the system into three dimensions.

However, FSAs extracted from neural networks rarely contain a minimal representation of the grammar learnt by the network. The states represented by the hidden layer tend to smear and overlap. If the network is being used to emulate an FSA, as in Das and Moser [1998], the locus of activation caused by the sequential input tends to shift for long strings. This results in an FSA that changes structure over time.

7.0 Results

Results in this section were obtained through implementation of the on-line clustering architecture outlined in section 5.0. The networks were trained with the Wales spoken language corpus, which consists of lexically tagged transcripts of conversations involving children of various ages. This corpus differs greatly from the commonly used test sets, such as the Tomita languages. These more common corpora are highly deterministic in nature, and have of course been constructed artificially. This allows relatively simple representations of their grammar by FSAs. Spoken language, on the other hand, has proven extremely difficult to represent with deterministic FSAs. The Wales corpus is not very predicable; bi-gram prediction results in a 40% correct prediction rate and a standard Elman recurrent architecture achieves a 45% correct predication rate.

7.1 Training

The data set used in obtaining these results consisted of lexically encoded conversations with six-year-old children, where the utterances not spoken by the test subject were removed. The data contained 23 lexical categories, and the network activations were reset at the beginning of each utterance. The network was trained to perform a one-step look-ahead task.

Networks were trained in three groups. The first group was trained with no on-line clustering, using the standard Elman recurrent network architecture. This group acted as a 'control' group. The second group was trained with the new on-line clustering architecture, using a value for $\boldsymbol{f}$ in eqtn (8) that ensured 'weak' clustering, where the hidden-layer activations were only weakly modified by the interpolation with their classified counterparts. This group was dubbed 'AVQ-lo'. The third group was likewise trained with the new architecture, but with a value for $\boldsymbol{f}$ that ensured 'strong' clustering, where the hidden-layer activations were pushed strongly towards the classified activations. This last group was dubbed 'AVQ-hi'.

All networks had the same basic topology: 23 neurons each in the input and output layers, and 16 neurons each in the hidden and context layers. The clustering method used for both on-line clustering as well as state extraction was the order-1 algorithm outlined in section 3.1.

7.2 Outcomes

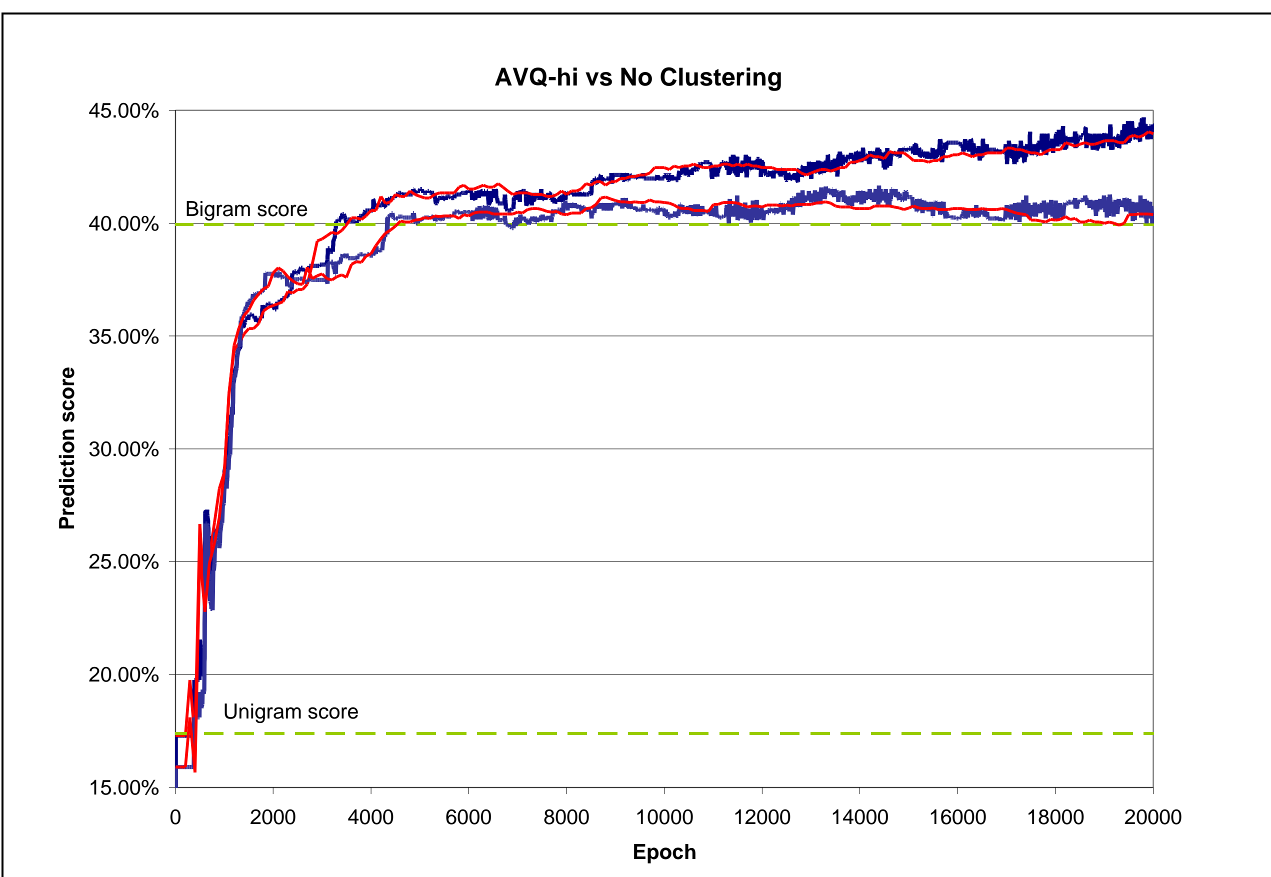


Figure 7-1 The result of training a single network with the on-line clustering architecture compared to a network trained without on-line clustering. The thick dark lines are the prediction scores for the AVQ-hi network on (top to bottom) training and test data. The thinner lines are the prediction scores for the networks trained without clustering, for the same input data.

Figure 7-1 shows the result of training with the on-line clustering architecture. The training and test results are comparable to the standard Elman networks. Averaged over a number of runs, the AVQ-hi networks predict slightly better than the standard nets on the test data.

Figure 7-2 shows the number of states required to extract an FSA from each of the network groups. The AVQ-hi group required half as many states as the networks trained without any clustering, and the resulting FSAs predicted more accurately on the one-step look-ahead task than the FSAs extracted from the control networks.

These results reproduce benefits identified by Das and Moser [1998]. However, the goal of our research was to advance this result further by using a more sophisticated clustering scheme. The PCA plot in Figure 6-1 at least partially shows that the 'natural' clusters formed by the hidden layer activations are not hyper-spherical, but are elongated. It was proposed that a hyper-ellipsoidal cluster would more naturally represent the states of the internal FSA. Therefore a network trained

with on-line clustering based on a second-order distance metric should produce a further reduction in size of the extracted FSA, while obtaining still better prediction scores.

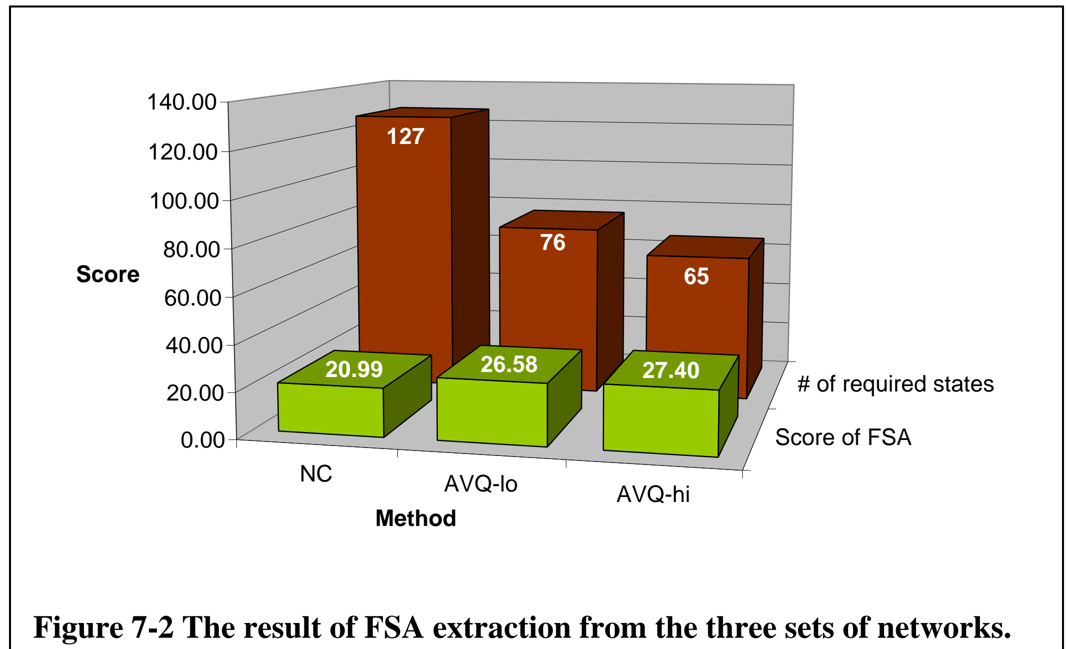


Figure 7-2 The result of FSA extraction from the three sets of networks.

The second-order distance metric outlined in section 2.2 was implemented to test this hypothesis. The details of this implementation are in section 9.3.12. A clustering algorithm was developed to use this metric, and was incorporated into tlearn in the same way as tlavq (section 9.2.11). This algorithm was tested on a small data set of features extracted from musical melodies [Towsey et. al. 2001, p63]. The resulting clusters were compared against a PCA plot, which showed that the clusters were elongated in the direction of the second principal component. This confirms that the clustering algorithm was identifying the most important features of the data.

This algorithm worked, but re-classified every data point several times for each new cluster generated. The algorithm scaled very badly to the large data sets used by the on-line clustering system, to the extent that results could not be obtained in reasonable lengths of time.

The algorithm in section 3.2 was an attempt to resolve this issue, by constructing an algorithm that would not require frequent re-classification of the entire data set. Unfortunately this algorithm was not working by the end of the project. The problem seems to be in the initial formation of clusters. When a cluster consists of a small number of points, the correlation matrix representation becomes badly skewed along the principal axis of the cluster. Various attempts were made to solve this problem, with limited success. Section 9.2.10 contains o2clust, the implementation of the algorithm in section 3.2. The notes for o2clust contain more detail about the attempted solutions and some suggestions for further modifications.

8.0 References

Das, S. & Moser, M. 1998, 'Dynamic On-Line Clustering and State Extraction: An Approach to Symbolic Learning', *Neural Networks*, vol. 11, no. 1, pp.53-64.

Elman, J. 1990, 'Finding Structure in Time', *Cognitive Science*, vol. 14, pp. 179-211.

Elman, J. 1999, *Tlearn Software Page* [Online]. Available: http://crl.ucsd.edu/innate/tlearn.html [2001, October 12].

Everitt, B. 1977, *Cluster Analysis*, Heinemann Educational Books, London.

Giles, C., Miller, C.B., Chen, H., Sun, G. & Lee, Y. 1992, 'Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, vol. 4, no. 4, pp. 393-405.

Giles, C.L. & Omlin, C.W. 1993a, 'Rule refinement with recurrent neural networks', *Proceedings of the IEEE International Conference on Neural Networks*, pp. 801-806, San Francisco.

Giles, C.L. & Omlin, C.W. 1993b, 'Extraction, insertion and refinement of symbolic rules in dynamically driven recurrent networks', *Connection Science*, vol. 5, no. 3 & 4, pp. 307-328.

Kreyzig, E. 1993, *Advanced Engineering Mathematics*, John Wiley & Sons Inc, Singapore.

Lipson, H. & Siegelmann, H. T. 1998, 'High Order Shape Neurons for Data Structure Decomposition', *NIPS Workshop: Hybrid Neural Symbolic Integration*, December 4th 1998.

Sayood, K. 1996, *Introduction to Data Compression*, Morgan Kaufmann Publishers, San Francisco.

Schellhammer, I., Diederich, J., Towsey, M. & Brugman, C. 1998, 'Knowledge extraction and recurrent neural networks: an analysis of an Elman network trained on a natural language learning task', *Proceedings of the Joint Conference on New Methods in Language Processing and Computational Natural Language Learning*, pp. 73-78.

Towsey, M. 1998, *The Use of Neural Networks in the Automated Analysis of the Electroencephalogram*, Ph.D. Thesis, University of Queensland, Brisbane, Australia.

Towsey, M., Brown, A., Wright, S., & Diederich, J. 2001, 'Towards Melodic Extension Using Genetic Algorithms', *Educational Technology & Society*, vol. 4, no. 2, pp. 54-65.

Zupan, J. 1982, *Clustering of Large Data Sets*, John Wiley & Sons Ltd., Chichester.

Part 2 – Software

9.0 Software

9.1 Software Table of Contents

9.1 Software Table of Contents 37
9.2 Applications and Utilities 39
9.2.1 tlearn 39
9.2.2 tlearn's Internal Structure 42
tlearn.c 42
parse.c 50
update.c 51
activate.c 52
compute.c 52
arrays.c 53
weights.c 54
subs.c 54
exp.c 55
9.2.3 dstat 56
test.pattern 56
test.tags 56
test.unigram 56
test.bigram 57
Makefile 57
dstat.c 57
9.2.4 SymStrip 64
Makefile 64
ui.h 64
ui.c 65
symcmds.h 70
SymCmds.c 71
SymSyms.c 73
SymOpts.c 76
SymStrip.h 81
SymStrip.c 82
9.2.5 MakeFSA 85
Makefile 85
MakeFSA.c 86
makefsa.help.make 94
9.2.6 pattern 96
Makefile 96
pattern.c 96
9.2.7 vector 100
example.txt 100
wales.claire.tags 100
Makefile 101
vector.c 101
9.2.8 tlbe 109
Makefile 109
tlbe.c 109
9.2.9 sclust 115
Makefile 115
sclust.help.make 115
sclust.c 116
9.2.10 o2clust 126
Makefile 127
o2clust.help.make 127
o2clust.c 127
9.2.11 tlavq 138
avq.h 139

avq.c 140
avq_subs.h 143
avq_subs.c 144
tlavq.c 148
weights.c 156
arrays.c 158
parse.c 161
activate.c 168
compute.c 171
subs.c 176
update.c 178
9.3 Source Code Modules 186
9.3.1 StdDefs 186
StdDefs.h 186
9.3.2 2darray 186
2darray.h 187
2darray.c 187
9.3.3 gauss 188
gauss.h 188
gauss2.c 188
9.3.4 RunAvg 189
RunAvg.h 190
RunAvg.c 190
9.3.5 htable 191
htable.h 191
htable.c 193
9.3.6 smdarray 197
smdarray.h 199
smdarray.c 200
9.3.7 TokenLst 204
TokenLst.h 204
TokenLst.c 205
9.3.8 TokScan 207
TokScan.h 207
TokScan.c 208
9.3.9 vector_utils 209
vector_utils.h 209
vector_utils.c 211
9.3.10 vector_read 215
vector_read.h 216
vector_read.c 216
9.3.11 cluster 217
cluster.h 218
cluster.c 219
9.3.12 corr_matrix 222
corr_matrix.h 222
corr_matrix.c 226

9.2 Applications and Utilities

9.2.1 tlearn

tlearn is a software package written by Jeff Elman and others. It simulates "layers" of neurons interacting with user-specified connections, and can perform training by back-propagation. This portion of the technical report describes the inner workings of tlearn and the components that make up the software. The intent is for the reader to be able to reverse-engineer the software in order to extend it.

An example of such an extension is **tlavq** (described in section 9.2.11), which extends **tlearn** by adding an on-line clustering module with associated command-line switches.

Networks modeled by tlearn

Neurons in the network are grouped into layers, and connections between these layers are specified by the user in the command file (which usually has a .cf extension). The default connectivity is for complete connections from one layer to another. This can be changed to copy-back connections for recurrent nets. The default neuron transfer function is a sigmoid, but can also be specified to be linear. Links are by default included in any training performed, but can be set to fixed untrainable weights if desired. Trained networks can be run through the network to extract their outputs as well as their hidden unit activations.

tlearn network configuration files

The structure of the neural network for simulation is defined at run-time using a configuration (.cf) file.

example.cf

`NODES:`	
`nodes = 12`	Specifies the number of neurons in the network. Inputs do **not** count as nodes.
`inputs = 4`	Specifies the number of inputs into the network.
`outputs = 4`	Specifies the number of outputs from the network. Output neurons are included in the node count.
`output nodes are 5-8`	Specifies which nodes are to be sampled for their outputs. (Also used for training)
`CONNECTIONS:`	
`groups = 0`	Specifies the number of groups of nodes. Groups are constrained to have the same weights on their connections. This is not terribly useful, and is usually 0.
`1-4 from i1-i4`	Specifies that nodes 1 through 4 are fully connected from inputs 1 through 4.
`5-8 from 1-4`	Specifies that the output nodes (5 through 8) are fully connected from hidden units 1 through 4.
`9-12 from 1-4 = 1.0 & 1.0 fixed one-to-one`	This line specifies that nodes 9 through 12 are "context" neurons. Their connections from nodes 1 through 4 are constrained to have weights of 1.0. Their activations are copied from nodes 1 through 4 on each sweep.
`SPECIAL:`	
`linear = 9-12`	Specifies that neurons 9 through 12 use a linear output

	function.
`selected = 1-4`	Specifies that when performing a run to probe the network's hidden units, the activations from nodes 1 through 4 will be extracted.

This configuration file describes the network shown below.

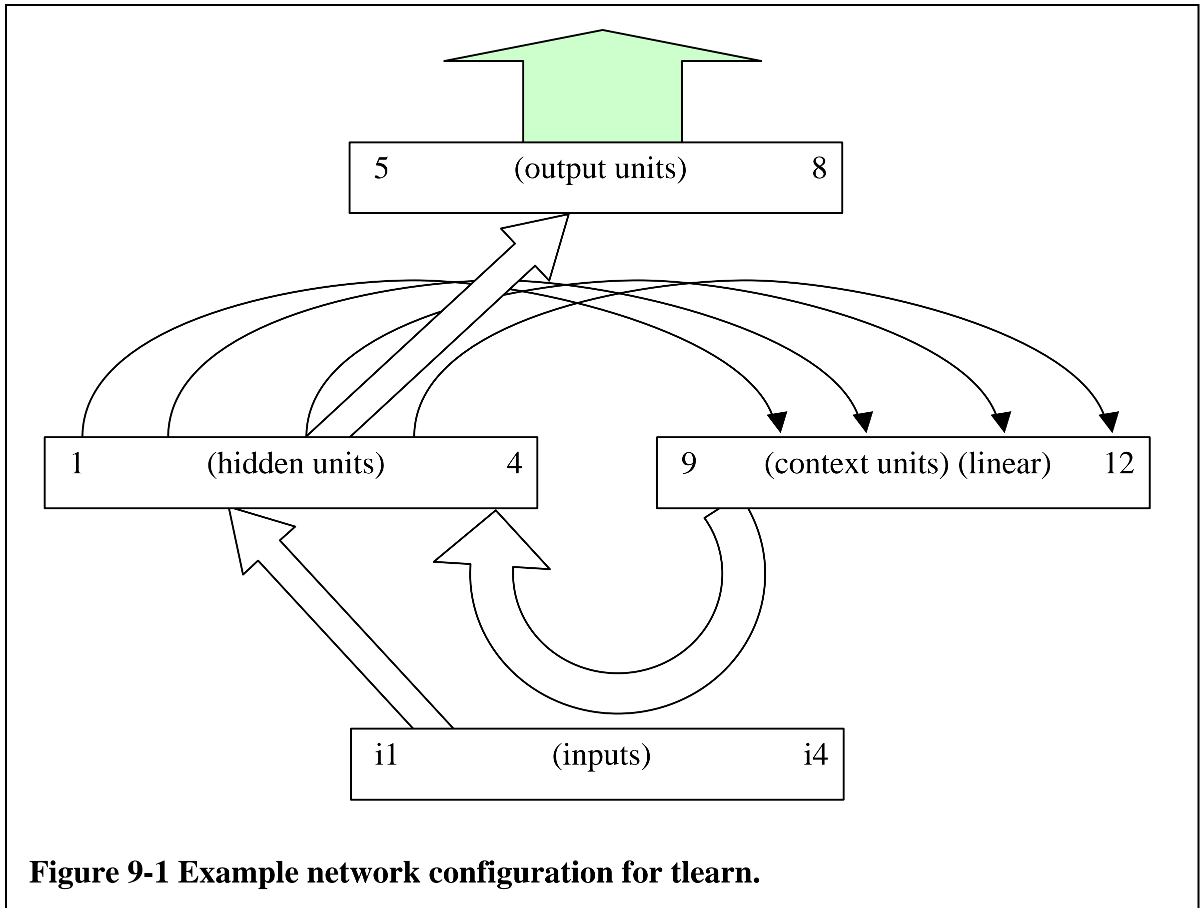


Figure 9-1 Example network configuration for tlearn.

For a more detailed description of the configuration options, see the tlearn manual. The next page is a quick reference guide to **tlearn**'s command-line switches.

tlearn

–f<fileroot>

–r<learning.rate>

–m<momentum>

–s<sweeps>

–l<old.weights.file>

–X (use.reset)

–M<RMSerr.target>

–E<RMSerr.checkpoint>

–C<weights.checkpoint>

–S<RNG.seed>

–V<Verify learning>

–R (randomise.input)

–P (probe.selected)

Figure 9-2 tlearn quick reference page.

9.2.2 tlearn's Internal Structure

tlearn is written in the Kernighan and Ritchie dialect of C. It has no function prototypes and uses a non-ANSI method for declaring function parameters, which can make understanding the source code difficult at times. This section contains the documentation resulting from the reverse-engineering process, including function prototypes and descriptions as well as the global variable definitions.

Source code files

tlearn.c	• Reads command-line switches • Opens configuration file • Loads weights from file (if specified) • Initialises biases • Main sweep loop
parse.c	• Reads and interprets configuration file • Sets up network configuration arrays
update.c	• Loads inputs and outputs from disk • Manages updating of targets and input values • Manages updating of weights (but not their desired values)
activate.c	• Calculates network activations
compute.c	• Calculates error measures • Calculates error signals and desired weight deltas • Determines weight updates required for learning
arrays.c	• Allocates arrays
weights.c	• Saves and loads weights to/from disk
subs.c	• Various misc. output functions • Network reset functions
exp.c	• (Separate program) creates a lookup table for exponential function

Source file module descriptions

tlearn.c

Structures

```
    struct   cf {           /* Connection information */
        int   con,           /* FLAG: This connection exists */
              fix,           /* FLAG: This connection has a fixed weight */
              num            /* Group number this connection belongs to */
              lim;           /* FLAG: This connection has weight limits imposed */
        float min,           /* Minimum allowed weight */
              max;           /* Maximum allowed weight */
    };
```

This structure is used in a two-dimensional array **cinfo**. Each cell holds the connection information between two nodes:

```
  struct cf    **cinfo[to][from].
```

High-dimensional arrays in tlearn are indexed separately along their dimensions. For example,

```
  struct cf    *ci = cinfo + to;
  *(ci + from) = data;
```

```
    struct   nf {           /* Node information */
```

```
        int     func,           /* Which output function this node uses */
                dela,           /* FLAG: This node has a delayed output */
                targ;           /* FLAG: This node is an output node (i.e. a target for learning */
    };
```

This structure is used in the array **ninfo**. Each cell holds the node information:

```
  struct nf     *ninfo[node].
```

Exported variables

```
    /* Node counts */
        int     nn,             /* Number of "real" nodes (i.e. not including inputs or bias nodes) */
                ni,             /* Number of input nodes */
                no,             /* Number of designated output nodes */
                nt,             /* Total number of nodes (nn + ni + 1) */
                np;             /* Index to first "real" node (ni + 1) */

    /* Information arrays */
        struct cf **cinfo;      /* 2D ARRAY[nn][nt]: Connection information between all nodes */
        struct nf *ninfo;       /* ARRAY[nn]: Node information for all nodes */

    /* Node flag arrays */
        int     *outputs,       /* ARRAY[no]: Indices of output nodes */
                *selects,       /* FLAG ARRAY[nn + 1]: This node is selected for probe output */
                *linput;        /* ARRAY[ni]: Localist input array */

    /* Network state arrays */
        float   *znew,          /* ARRAY[nt]: Activations of nodes at time t + 1 */
                *zold,          /* ARRAY[nt]: Activations of nodes at time t */
                *zmem,          /* ARRAY[nt]: Saved activations of nodes at time t */
                **wt,           /* 2D ARRAY[nn][nt]: Weight of connection [to][from] */
                **dwt,          /* 2D ARRAY[nn][nt]: Delta weight to rectify error signal */
                **winc,         /* 2D ARRAY[nn][nt]: Actual desired weight increase for learning */
                *target,        /* ARRAY[no]: Target output values */
                *error,         /* ARRAY[nn]: Error signals (output – target) */
                ***pnew,        /* 3D ARRAY[nn][nt][nn]: P-variable at time t+1 */
                ***pold;        /* 3D ARRAY[nn][nt][nn]: P-variable at time t */

    /* Network parameters */
        float   rate,           /* Learning rate (default: 0.1) */
            momentum,           /* Momentum for learning (default: 0.0) */
            weight_limit,       /* Bound for random weight initialisation limit (default: 1.0) */
            criterion,          /* RMS error target for end of learning (default: 0.0) */
            init_bias;          /* Offset for bias weight initialisation (default: 0.0) */
        long    sweep,          /* Current sweep */
                tsweeps,        /* Total number of sweeps on this network */
                report;         /* Output RMS error every 'report' sweeps (default: 0.0) */
        int     ngroups;        /* Number of groups */

    /* Network flags */
        int     backprop,       /* FLAG: Use standard back-propagation? (default: YES) */
                teacher,        /* FLAG: Feed back targets during learning? (default: NO) */
                localist,       /* FLAG: Use localist-type input values? */
                randomly,       /* FLAG: Present inputs in random order? (default: NO) */
```

```
            limits,         /* FLAG: Limit weight values? (default: NO) */
            ce;             /* FLAG: Use cross-entropy error collection? (default: NO) */

    /* Files */
        char  root[128],    /* Root portion of filename (i.e. 'root'.data, .teach, .cf, .reset, etc) */
              loadfile[128]; /* Filename of saved weights file to initialise network with */
        FILE *cfp;          /* File pointer for .cf file */
```

Function prototypes

```
int main(IN int argc, IN char **argv);
/* -- Function main
 * Pre: TRUE
 * Post: TRUE
 * Purpose: Initialise network, main sweep loop
 */

void usage(void);
/* -- Function usage
 * Pre: TRUE
 * Post: The command-line usage has been output to stderr
 */

void intr(IN int sig);
/* -- Function intr
 * Pre: sig is the received termination signal form the OS
 * Post: The current set of weights has been saved and tlearn has exit
 */
```

Annotated source

```
/* tlearn.c - simulator for arbitrary networks with time-ordered input */

/*------------------------------------------------------------------------

This program simulates learning in a neural network using either
the classical back-propagation learning algorithm or a slightly
modified form derived in Williams and Zipser, "A Learning Algo-
rithm for Continually Running Fully Recurrent Networks."  The
input is a sequence of vectors of (ascii) floating point numbers
contained in a ".data" file.  The target outputs are a set of
time-stamped vectors of (ascii) floating point numbers (including
optional "don't care" values) in a ".teach" file.  The network
configuration is defined in a ".cf" file documented in tlearn.man.

------------------------------------------------------------------------*/

#include <math.h>
#include <stdio.h>
#include <signal.h>
#ifdef ibmpc
#include "strings.h"
#include <fcntl.h>
#else
#include <string.h>
#include <sys/file.h>
#endif
#include <sys/types.h>
#include <sys/stat.h>

#ifdef ibmpc
#define random(x) rand(x)
#define srandom(x) srand(x)
#endif
```

Node count variables

```
int   nn;      /* number of nodes */
int   ni;      /* number of inputs */
int   no;      /* number of outputs */
int   nt;      /* nn + ni + 1 */
int   np;      /* ni + 1 */
```

Information structures

```
struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight-limits flag */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};
```

Information arrays

```
struct   cf **cinfo; /* (nn x nt) connection info */
struct   nf *ninfo;     /* (nn) node activation function info */
```

Node flag arrays

```
int   *outputs;   /* (no) indices of output nodes */
int   *selects;   /* (nn+1) nodes selected for probe printout */
int   *linput; /* (ni) localist input array */
```

Activations, weights, target, error signal and p-variable arrays

```
float *znew;      /* (nt) inputs and activations at time t+1 */
float *zold;      /* (nt) inputs and activations at time t */
float *zmem;      /* (nt) inputs and activations at time t */
float **wt;    /* (nn x nt) weight TO node i FROM node j*/
float **dwt;      /* (nn x nt) delta weight at time t */
float **winc;     /* (nn x nt) accumulated weight increment*/
float *target; /* (no) output target values */
float *error;     /* (nn) error = (output - target) values */
float ***pnew; /* (nn x nt x nn) p-variable at time t+1 */
float ***pold; /* (nn x nt x nn) p-variable at time t */
```

Network parameters

```
float rate = .1;  /* learning rate */
float momentum = 0.; /* momentum */
float weight_limit = 1.;   /* bound for random weight init */
float criterion = 0.;   /* exit program when rms error is less than this */
float init_bias = 0.;   /* possible offset for initial output biases */
```

```
long  sweep = 0;  /* current sweep */
long  tsweeps = 0;   /* total sweeps to date */
long  report = 0; /* output rms error every "report" sweeps */

int   ngroups = 0;   /* number of groups */
```

```
int   backprop = 1;  /* flag for standard back propagation (the default) */
int   teacher = 0;   /* flag for feeding back targets */
int   localist = 0;  /* flag for speed-up with localist inputs */
int   randomly = 0;  /* flag for presenting inputs in random order */
int   limits = 0; /* flag for limited weights */
int   ce = 0;     /* flag for cross_entropy */
```

Network flags

```
char  root[128];  /* root filename for .cf, .data, .teach, etc.*/
char  loadfile[128]; /* filename for weightfile to be read in */

FILE  *cfp;    /* file pointer for .cf file */
```

Files

```
void intr();

extern int load_wts();
extern int save_wts();
extern int act_nds();


main(argc,argv)
   int   argc;
   char  **argv;
```

Function **main**

```
{

   FILE  *fopen();
   extern   char *optarg;
   extern   float rans();
   extern   long time();
/* extern   int intr(); */


   long  nsweeps = 0;   /* number of sweeps to run for */
   long  ttime = 0;  /* number of sweeps since time = 0 */
   long  utime = 0;  /* number of sweeps since last update_weights */
   long  tmax = 0;   /* maximum number of sweeps (given in .data) */
   long  umax = 0;   /* update weights every umax sweeps */
   long  rtime = 0;  /* number of sweeps since last report */
   long  check = 0;  /* output weights every "check" sweeps */
   long  ctime = 0;  /* number of sweeps since last check */
```

Time variables

ttime: sweeps since start of .data file

tmax: number of elements in .data

```
   int   c;
   int   i;
   int   j;
   int   k;
   int   nticks = 1; /* number of internal clock ticks per input */
   int   ticks = 0;  /* counter for ticks */
   int   learning = 1;  /* flag for learning */
   int   reset = 0;  /* flag for resetting net */
   int   verify = 0; /* flag for printing output values */
   int   probe = 0;  /* flag for printing selected node values */
   int   command = 1;   /* flag for writing to .cmd file */
   int   loadflag = 0;  /* flag for loading initial weights from file */
   int   iflag = 0;  /* flag for -I */
   int   tflag = 0;  /* flag for -T */
   int   rflag = 0;  /* flag for -x */
   int   seed = 0;   /* seed for random() */
```

Flags and indices

```
   float err = 0.;   /* cumulative ss error */
   float ce_err = 0.;   /* cumulate cross_entropy error */
```

Measured error values

```
   float *w;
   float *wi;
   float *dw;
   float *pn;
   float *po;
```

Weight and p-variable array indices

```
   struct   cf *ci;
```

Connection information array index

```
   char  cmdfile[128];  /* filename for logging runs of program */
   char  cfile[128]; /* filename for .cf file */
```

.cf and .cmd filenames

```
   FILE  *cmdfp;
```

.cmd file pointer

```
   signal(SIGINT, intr);
#ifndef ibmpc
   signal(SIGHUP, intr);
```

Set up signal handlers

```
   signal(SIGQUIT, intr);
   signal(SIGKILL, intr);
#endif

#ifndef ibmpc
   exp_init();
#endif

   root[0] = 0;

   while ((c = getopt(argc, argv, "f:hil:m:n:r:s:tC:E:ILM:PRS:TU:VXB:H:")) != EOF) {
      switch (c) {
         case 'C':
            check = (long) atol(optarg);
            ctime = check;
            break;
         case 'f':
            strcpy(root,optarg);
            break;
         case 'i':
            command = 0;
            break;
         case 'l':
            loadflag = 1;
            strcpy(loadfile,optarg);
            break;
         case 'm':
            momentum = (float) atof(optarg);
            break;
         case 'n':
            nticks = (int) atoi(optarg);
            break;
         case 'P':
            probe = 1;
            learning = 0;
            break;
         case 'r':
            rate = (float) atof(optarg);
            break;
         case 's':
            nsweeps = (long) atol(optarg);
            break;
         case 't':
            teacher = 1;
            break;
         case 'L':
            backprop = 0;
            break;
         case 'V':
            verify = 1;
            learning = 0;
            break;
         case 'X':
            rflag = 1;
            break;
         case 'E':
            report = (long) atol(optarg);
            break;
         case 'I':
            iflag = 1;
            break;
         case 'M':
            criterion = (float) atof(optarg);
            break;
         case 'R':
            randomly = 1;
            break;
         case 'S':
            seed = atoi(optarg);
            break;
         case 'T':
            tflag = 1;
            break;
         case 'U':
            umax = atol(optarg);
            break;
         case 'B':
            init_bias = atof(optarg);
            break;
         /*
          * if == 1, use cross-entropy as error;
          * if == 2, also collect cross-entropy stats.
```

Get options

from command line

```
        */
       case 'H':
          ce = atoi(optarg);
          break;
       case '?':
       case 'h':
       default:
          usage();
          exit(2);
       break;
    }
 }

 if (nsweeps == 0){
    perror("ERROR: No -s specified");
    exit(1);
 }
```

Get options from command line

```
 /* open files */

 if (root[0] == 0){
    perror("ERROR: No fileroot specified");
    exit(1);
 }

 if (command){
    sprintf(cmdfile, "%s.cmd", root);
    cmdfp = fopen(cmdfile, "a");
    if (cmdfp == NULL) {
       perror("ERROR: Can't open .cmd file");
       exit(1);
    }
    for (i = 1; i < argc; i++)
       fprintf(cmdfp,"%s ",argv[i]);
    fprintf(cmdfp,"\n");
    fflush(cmdfp);
 }

 sprintf(cfile, "%s.cf", root);
 cfp = fopen(cfile, "r");
 if (cfp == NULL) {
    perror("ERROR: Can't open .cf file");
    exit(1);
 }
```

Open files

```
 get_nodes();
 make_arrays();
 get_outputs();
 get_connections();
 get_special();
```

Read network configuration and set up network arrays

```
 if (!seed)
    time(&seed);
 srandom(seed);
```

Initialise random number generator

```
 if (loadflag)
    load_wts();
 else {
    for (i = 0; i < nn; i++){
       w = *(wt + i);
       dw = *(dwt+ i);
       wi = *(winc+ i);
       ci = *(cinfo+ i);
       for (j = 0; j < nt; j++, ci++, w++, wi++, dw++){
          if (ci->con)
             *w = rans(weight_limit);
          else
             *w = 0.;
          *wi = 0.;
          *dw = 0.;
       }
    }
    /*
     * If init_bias, then we want to set initial biases
     * to (*only*) output units to a random negative number.
     * We index into the **wt to find the section of receiver
     * weights for each output node.  The first weight in each
     * section is for unit 0 (bias), so no further indexing needed.
     */
    for (i = 0; i < no; i++){
       w = *(wt + outputs[i] - 1);
       ci = *(cinfo + outputs[i] - 1);
```

Initialise network weights either to random values or to a saved state, if **loadflag** is set.

```
      if (ci->con)
         *w = init_bias + rans(.1);
      else
         *w = 0.;
   }
}
zold[0] = znew[0] = 1.;
for (i = 1; i < nt; i++)
   zold[i] = znew[i] = 0.;
if (backprop == 0){
   make_parrays();
   for (i = 0; i < nn; i++){
      for (j = 0; j < nt; j++){
         po = *(*(pold + i) + j);
         pn = *(*(pnew + i) + j);
         for (k = 0; k < nn; k++, po++, pn++){
            *po = 0.;
            *pn = 0.;
         }
      }
   }
}
```

Initialising weights to random values

```
nsweeps += tsweeps;
for (sweep = tsweeps; sweep < nsweeps; sweep++){
```

Main sweep loop start

```
   for (ticks = 0; ticks < nticks; ticks++){
```

Tick loop start

```
      update_reset(ttime,ticks,rflag,&tmax,&reset);
```

Retrieve reset flag from .reset file

```
      if (reset){
         if (backprop == 0)
            reset_network(zold,znew,pold,pnew);
         else
            reset_bp_net(zold,znew);
      }
```

Reset entire network if required (if **reset** flag is set)

```
      update_inputs(zold,ticks,iflag,&tmax,&linput);
```

Get input values from .data file

```
      if (learning || teacher || (report != 0))
         update_targets(target,ttime,ticks,tflag,&tmax);
```

Update target outputs if learning

```
      act_nds(zold,zmem,znew,wt,linput,target);
```

Update network activations

```
      if (learning || (report != 0))
         comp_errors(zold,target,error,&err,&ce_err);
```

Calculate error measures if required

```
      if (learning && (backprop == 0))
         comp_deltas(pold,pnew,wt,dwt,zold,znew,error);
      if (learning && (backprop == 1))
         comp_backprop(wt,dwt,zold,zmem,target,error,linput);
```

Calculate weight changes for learning if required

```
      if (probe)
         print_nodes(zold);
   }
```

Output selected nodes activations if required

```
   if (verify)
      print_output(zold);
```

Output output nodes activations if required

```
   if (report && (++rtime >= report)){
      rtime = 0;
      if (ce == 2)
         print_error(&ce_err);
      else
         print_error(&err);
   }
```

Report the error measures if required

```
   if (check && (++ctime >= check)){
      ctime = 0;
      save_wts();
   }
```

Save current weights to a checkpoint file if required

```
   if (++ttime >= tmax)
      ttime = 0;
```

Next data value, check for roll-over to start of .data file

```
   if (++utime >= umax){
      utime = 0;
      update_weights(wt,dwt,winc);
   }
```

Next update sweep, check whether to update weights and do so if required

```
}
```

If training occurred, save the weights.

```
   if (learning)
      save_wts();
   exit(0);

}
```

Function **usage**

```
usage() {
  fprintf(stderr, "\n");
  fprintf(stderr, "-f fileroot:\tspecify fileroot <always required>\n");
  fprintf(stderr, "-l weightfile:\tload in weightfile\n");
  fprintf(stderr, "\n");
  fprintf(stderr, "-s #:\trun for # sweeps <always required>\n");
  fprintf(stderr, "-r #:\tset learning rate to # (between 0. and 1.) [0.1]\n");
  fprintf(stderr, "-m #:\tset momentum to # (between 0. and 1.) [0.0]\n");
  fprintf(stderr, "-n #:\t# of clock ticks per input vector [1]\n");
  fprintf(stderr, "-t:\tfeedback teacher values in place of outputs\n");
  fprintf(stderr, "\n");
  fprintf(stderr, "-S #:\tseed for random number generator [random]\n");
  fprintf(stderr, "-U #:\tupdate weights every # sweeps [1]\n");
  fprintf(stderr, "-E #:\trecord rms error in .err file every # sweeps [0]\n");
  fprintf(stderr, "-C #:\tcheckpoint weights file every # sweeps [0]\n");
  fprintf(stderr, "-M #:\texit program when rms error is less than # [0.0]\n");
  fprintf(stderr, "-X:\tuse auxiliary .reset file\n");
  fprintf(stderr, "-P:\tprobe selected nodes on each sweep (no learning)\n");
  fprintf(stderr, "-V:\tverify outputs on each sweep (no learning)\n");
  fprintf(stderr, "-R:\tpresent input patterns in random order\n");
  fprintf(stderr, "-I:\tignore input values during extra clock ticks\n");
  fprintf(stderr, "-T:\tignore target values during extra clock ticks\n");
  fprintf(stderr, "-L:\tuse RTRL temporally recurrent learning\n");
  fprintf(stderr, "-B #:\toffset for offset biasi initialization\n");
  fprintf(stderr, "\n");
}
```

Function **intr**

```
void
intr(sig)
   int   sig;
{
   save_wts();
   exit(sig);
}
```

parse.c

Function prototypes

```
void get_nodes(void);
/* -- Function get_nodes
 * Pre: "root.cf" file has been opened
 *      cfp is a global file pointer to the configuration file
 *      The stream is at the start of the file
 *Post: Sets nn, ni, no, nt and np according to configuration file
 */

void get_outputs(void);
/* -- Function get_outputs
 * Pre: "root.cf" file has been opened
 *      cfp is a global file pointer to the configuration file
 *      get_nodes has been called
 * Post: Reads "outputs" line from "NODES" section of configuration file
 *       Fills outputs[] array according to configuration file
 */

void get_connections(void);
/* -- Function get_connections
 * Pre: "root.cf" file has been opened
 *      cfp is a global file pointer to the configuration file
 *      get_outputs has been called
 * Post: Reads "CONNECTIONS" section of configuration file
 *       Fills cinfo[] structure array according to network layout
 */

void get_special(void);
/* -- Function get_special
 * Pre: "root.cf" file has been opened
```

```
 *      cfp is a global file pointer to the configuration file
 *      get_connections has been called
 * Post: Reads "SPECIAL" section of configuration file
 *       Sets weight_limit global variable
 *       Sets selects[] array for probing hidden nodes
 *       Sets flags in ninfo[] structure array
 */

void get_str(IN FILE *fp, OUT char buf[], IN char str[]);
/* -- Functon get_str
 * Pre: fp is a file pointer to a file opened for reading
 *      buf[] is a pre-allocated string buffer to read data into
 *      str[] is the whitespace character expected after the next string
 * Post: A whitespace-terminated string was read into buf
 */

void get_nums(IN char str[], IN int nv, IN int offset, OUT int vec[]);
/* -- Function get_nums
 * Pre: str[] contains a string to interpret as a numeric set of numbers
 *      nv is the length of vec
 *      offset is the offset to "real" nodes in the node arrays
 *      vec[] is a pre-allocated array of flags for selection
 * Post: The numeric set in str was interpreted
 *       vec[] is a set of flags.  The cell i was set to YES if i was in
 *        the set given in str[]
 * Notes: Numeric sets are of the form a,b,c-d,e,… etc
 */

void parse_err(void);
/* -- Function parse_err
 * Pre: TRUE
 * Post: An error was printed to stderr
 */
```

update.c

Function prototypes

```
void update_inputs(  OUT float aold[], IN int tick, IN int flag,
                     OUT long *maxtime, OUT int *local[]);
/* -- Function update_inputs
 * Pre: the global variable root[] contains the root filename
 *      aold[] is the current activations in which to place the new
 *       input values
 *      tick is the current tick in the current sweep
 *      flag specifies that the inputs should be set to zero after
 *       the first tick in a sweep (-I switch)
 *      maxtime is the number of input vectors in the .data file
 *      local is the input array for localist inputs
 * Post: The first time this function is called, it loads the .data file
 *        and sets maxtime.  On subsequest executions, it loads the next
 *        input vector from the sequence into either aold[] or local[],
 *        depending on the global flag localist.
 */

void update_targets( OUT float atarget[], IN long time, IN int tick,
                     IN int flag, IN long *maxtime);
/* -- Function update_targets
 * Pre: the global variable root[] contains the root filename
 *      atarget[] is the pre-allocated array of targets to update
 *      time is the index of the current input vector in the .data file
 *      tick is the current tick in the current sweep
 *      flag specifies that the targets should be set to zero after
 *       the first tick in a sweep (-T switch)
 *      maxtime is the number of input vectors in the .data file
```

```
 * Post: The first time this function is called, it loads the .teach file.
 *        On subsequent executions, it updates the targets in atarget[] to
 *        match the current training vector represented by time.
 */

void update_reset( IN long time, IN int tick, IN int flag,
                   IN long *maxtime, OUT int *now);
/* -- Function update_reset
 * Pre: the global variable root[] contains the root filename
 *      time is the index of the current input vector in the .data file
 *      tick is the current tick in the current sweep
 *      flag specifies that resets should be loaded from the .reset file
 *       (-X switch)
 *      maxtime si the number of input vectors in the .data file
 * Post: The first time this funciton is called, it loads the .reset
 *        file if required.  On subsequent executions, it updates the flag
 *        now to indicate if the network activations should be reset
 *        immediately before this learning sweep.
 */

void update_weights(  IN OUT float awt[][], IN OUT float adwt[][],
                      OUT float awinc[][]);
/* -- Function update_weights
 * Pre: awt[][] contains the current array of network weights
 *      adwt[][] contains the desired weight deltas for this learning sweep
 *      awinc[][] is a pre-allocated 2D ARRAY[nn][nt]
 * Post: Non-fixed weights have been updated according to the global
 *        learning rate rate and the global learning momentum momentum.
 *       Group weights have been averaged together.
 *       Limited weights have had their limits enforced.
 *       winc[][] contains the actual weight changes
 *       awt[][] contains the new weights
 *       adwt[][] has been zeroed
 */
```

activate.c

Function prototypes

```
void act_nds(  IN float aold[], OUT float amem[], OUT float anew[],
               IN float awt[][], IN int *local, IN float atarget[]);
/* -- Function act_nds
 * Pre: aold[] contains the network activations for the previous time step
 *      amem[] is an allocated ARRAY[nt]
 *      anew[] is an allocated ARRAY[nt]
 *      awt[][] contains the current network weights
 *      local is a FLAG: the input is in localist format
 *      atarget[] has been loaded with the current learning targets
 * Post: All network activations have been updated and placed in anew[]
 *       aold[] has been copied to amem[]
 *       learning targets have been fed back if the global flag teacher is
 *        set (-t switch)
 */
```

compute.c

Function prototypes

```
void comp_errors(  INT float aold[], INT float atarget[],
                   OUT float aerror[], IN OUT float *e,
                   IN OUT float *ce_e);
/* -- Function comp_errors
 * Pre: aold[] contains the current network activations
 *      atarget[] contains the current learning targets
 *      aerror[] is an allocated ARRAY[nn]
```

```
 *      e contains the accumulated error
 *      ce_e contains the accumulated cross-entropy error
 * Post: The error signals from atarget to aold have been calculated,
 *        and placed in aerror
 *       Either e has been updated or ce_e has been updated, depending
 *        on the global flag ce
 */

void comp_deltas(  IN OUT float apold[][][], OUT float apnew[][][],
                   IN float awt[][], IN OUT float adwt[][],
                   IN float aold[], IN float anew[], IN float aerror[]);
/* -- Function comp_deltas
 * Pre: apold[][][] contains the current p-variables for the network
 *      apnew[][][] is a pre-allocated 3D ARRAY[nn][nt][nn]
 *      awt[][] contains the current network weights
 *      adwt[][] contains the accumulated weight deltas
 *      aold[] contains the previous network activations
 *      anew[] contains the new network activations
 *      aerror[] contains the calculated error signals for the network
 * Post: apold[][][] has been updated
 *       apnew[][][] has been updated
 *       adwt[][] contains the updated weight deltas
 * Note: I don't really know what this function does.  Is it part of the
 *        Williams-Zipser algorithm for temporally recurrent learning?
 */

void comp_backprop(  IN float awt[][], IN OUT float adwt[][],
                     IN float aold[], IN float amem[],
                     IN float atarget[], OUT float aerror[],
                     IN int *local);
/* -- Function comp_backprop
 * Pre: awt[][] contains the current network weights
 *      adwt[][] contains the accumulated weight deltas
 *      aold[] contains the current network activations
 *      amem[] contains the previous network activations
 *      atarget[] contains the current learning targets
 *      aerror[] is a pre-allocated ARRAY[nn]
 *      local is a FLAG indicating that the inputs are in localist format
 * Post: The error signals from atarget[] to aold[] have been calculatd and
 *        placed in aerror[]
 *       The back-propagation learning algorithm has been used to
 *        calculate the desired weight deltas, which have been placed
 *        in adwt[][]
 */
```

arrays.c

Function prototypes

```
void make_arrays(void);
/* -- Function make_arrays
 * Pre: The network configuration file has been parsed
 * Post These global arrays have been allocated:
 *       zold[nt], zmem[nt], znew[nt]
 *       target[no], error[nn], outputs[no]
 *       selects[nt], linput[ni],
 *       wt[nn][nt], dwt[nn][nt], winc[nn][nt],
 *       cinfo[nn][nt], ninfo[nn]
 */

void make_parrays(void);
/* -- Function make_parrays
 * Pre: The network configuration file has been parsed
 * Post: These global arrats have been allocated:
 *        pold[nn][nt][nn], pnew[nn][nt][nn]
```

```
    */
```

weights.c

Function prototypes

```
void save_weights(void);
/* -- Function save_weights
 * Pre: The global variable root[] contains the root filename
 *      The global array wt[][] contains the network weights
 *      The global varibale sweep contains the current sweep number
 * Post: The network weights were saved to a file "root[].sweep.wts",
 *        in a manner that can be loaded by load_wts
 */

void load_wts(void);
/* -- Function load_wts
 * Pre: The global variable loadfile[] contains the filename of the saved
 *       state to load
 *      The global array wt[][] has been allocated
 * Post: The saved weights were loaded into wt[][]
 */
```

subs.c

Exported variables

/* Exp function lookup table */
float *exp_array; /* table for lookup */

Function prototypes

```
float rans(IN float w);
/* -- Function rans
 * Pre: w contains a range to generate random numbers: [-w..w]
 * Post: A random number in the range [-w..w] was returned
 */

void exp_init(void);
/* -- Function exp_init
 * Pre: TRUE
 * Post: The global array exp_table[] was allocated
 *       The data from the exp lookup file was read into exp_table[]
 */

void print_nodes(IN float aold[]);
/* -- Function print_nodes
 * Pre: aold[] contains the current network activations
 * Post: The network activations have been written to stdout
 */

void print_output(IN float aold[]);
/* -- Function print_output
 * Pre: aold[] contains the current network activations
 * post: The network outputs have been written to stdout
 */

void print_error(IN float *e);
/* -- Function print_error
 * Pre: The global variable root[] contains the root filename
 *      e contains the current accumulated error
 * Post: The first time this functon is called, it opens and initialises
 *        the error output file "root[].err".  On subsequent executions,
 *        it calculates either the RMS error or the cross-entropy error,
 *        depending on the global flag ce, and writes this error to the
```

```
 *        file.
 */

void reset_network(OUT float aold[], OUT float anew[],
                   OUT float apold[][][], OUT float apnew[][][]);
/* -- Function reset_network
 * Pre: aold[][] is a pre-allocated ARRAY[nn]
 *      anew[][]is a pre-allocated ARRAY[nn]
 *      apold[][][] is a pre-allocated 3D ARRAY[nn][nt][nn]
 *      apnew[][][]is a pre-allocated 3D ARRAY[nn][nt][nn]
 * Post: All cells in all parameter arrays have been zeroed
 */

void reset_bp_net(OUT float aold[], OUT float anew[]);
/* -- Function reset_bp_net
 * Pre: aold[] is a pre-alocated ARRAY[nn]
 *      anew[]is a pre-alocated ARRAY[nn]
 * Post: All cells in all parameter arrays have been zeroed
 */
```

exp.c

This stand-alone program generates a lookup table for the **exp** function, and writes it to **stdout**. When piped to a file, this is read by **init_exp** in *subs.c* to accelerate tlearn's calculation of network activations.

9.2.3 dstat

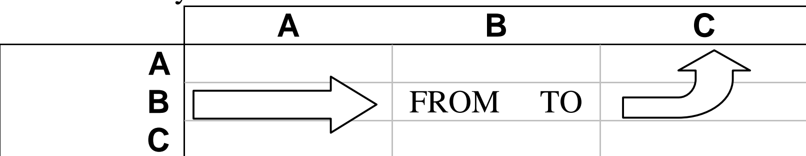
Application purpose

dstat will extract the uni-gram and bi-gram statistics for a data set, based on a {name}.pattern file. The .pattern file is in the format:

```
<pattern num -- zero origin> <predecessor> <successor>
<pattern num -- zero origin> <predecessor> <successor>
.
.
.
<EOF>
```

The predecessor and successor fields are numerical input identifiers. dstat is also supplied a "tags" file in order to associate the numerical identifiers with textual tags. When the data set statistics are output, these textual tags will be used to ease interpretation.

The transition arrays are formatted:



Usage

```
*** DataStat ver 0.22 -- Calculates dataset statistics
Usage: dstat {tagsFile} {patternFile} {outputBase}
Where: {tagsFile}    -- tags representing input lines
       {patternFile} -- file to specify output targets
       {outputBase}  -- base name of files to write output to
                        writes to {base}.unigram, {base}.bigram
                        (specify '-' for stdout)
For detailed info, type dstat --help
```

test.pattern

```
0 1 1
1 1 3
2 3 2
3 2 4
4 4 3
```

test.tags

```
A
B
C
D
```

The statistics are extracted by executing the command:

```
dstat test.tags test.pattern test
```

```
- DataStat ver 0.22
 | Pattern file (inputs, targets) [test.pattern]
 +-> Output files [test].unigram, .bigram
Read 4 tags (4 outputs)
```

test.unigram

```
---- Unigram stats
 4 categories
 Unigram score: 50.00%
(category) (count) (percentage)
        A       1       16.67%
        B       1       16.67%
        C       3       50.00%
        D       1       16.67%
------------------------------
                6      100.00%
```

test.bigram

```
---- Bigram stats
 4 categories
 Bigram score: 83.33%
 - - - - - - - - - - - - - - - - - - - - - - - - -
                   A           B           C           D
        A          1           0           1           0
        B          0           0           0           1
        C          0           1           0           0
        D          0           0           2           0
------------------------------
   6 patterns total


                   A           B           C           D
        A      16.67%       0.00%      16.67%       0.00%
        B       0.00%       0.00%       0.00%      16.67%
        C       0.00%      16.67%       0.00%       0.00%
        D       0.00%       0.00%      33.33%       0.00%
------------------------------
   100% total




Most common transitions
------------------------------
        A          C
        B          D
        C          B
        D          C
------------------------------
```

Modules used

StdDefs, TokenLst, TokScan

Source code

Makefile

```
OBJFILES=tokenlst.o tokscan.o dstat.o
TARGET=dstat
VERSION=022
DISTFILES=tokenlst.c tokenlst.h tokscan.c tokscan.h dstat.c ${TARGET}
HELPFILE=${TARGET}.help

CC=gcc -g

CFLAGS=
LFALGS=

${TARGET}: ${OBJFILES}
   ${CC} -o ${TARGET} ${OBJFILES}

clean:
   rm -f *.o ${TARGET}

dist: ${DISTFILES}
   tar cvf ${TARGET}_v${VERSION}.tar ${DISTFILES}
   gzip ${TARGET}_v${VERSION}.tar

help: ${HELPFILE} ${TARGET}

${HELPFILE}:
   ${TARGET} --help

.c.o:
   ${CC} ${CFLAGS} ${LFLAGS} -c $*.c
```

dstat.c

```
/* Dataset statistics -- uses .tags and .pattern file
 *                         calculats unigram and bigram stats
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG August 1999
```

```
 * Date: 19th August, 1999
 * Modified: 12th October, 1999
 * Version: 0.22
 */

/* --- Includes --- */

#include <stdio.h>
#include <string.h>
#include <unistd.h>     /* for exit() */
#include <stdlib.h>
#include "tokenlst.h"
#include "tokscan.h"
#include "stddefs.h"


/* --- Defines --- */

#define VERSION   "0.22"
#define HELPFILE "dstat.help"

#define  APPNAME     0
#define  TAGSFILE 1
#define PATTERNFILE  2
#define  OUTPUTBASE  3
#define  NUMARGS     4

#define  MAXTAGLENGTH   10
#define  MAX_STRING  512


/* --- Helper functions' definitions --- */

void Usage(FILE *output, char *appName);
void ProcessCommandLine(int argc, char *argv[], FILE **tagsFile, FILE ** patternFile, FILE
**unigramFile, FILE **bigramFile);
void Abort(int retnum);
void Help(char *appName);
void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs);
tokenList *ReadWholeFile(FILE *input);
void CloseFiles(FILE *tagsFile, FILE *patternFile, FILE *unigramFile, FILE *bigramFile);
void DoUnigramStats(FILE *patternFile, FILE *outputFile, int numOutputs, char *tagsArray[]);
void ReadPattern(FILE *patternFile, int *predecessor, int *successor, bool *inFile);
void OutputUnigramStats(char *tagsArray[], long outputCounts[], FILE *outputFile, int numOutputs,
int numPatterns, double unigramScore);
void DoBigramStats(FILE *patternFile, FILE *outputFile, int numOutputs, char *tagsArray[]);
void OutputBigramStats(char *tagsArray[], long **bigramCounts, int *transitions, FILE *outputFile,
int numOutputs, int numPatterns, double bigramScore);
bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize);
int MaxLongArray(long array[], int numElements);


/* --- Main --- */

int main(int argc, char *argv[])
{
   FILE  *tagsFile, *patternFile, *unigramFile, *bigramFile;
   char  **tagsArray;
   int   numOutputs;

   ProcessCommandLine(argc, argv, &tagsFile, &patternFile, &unigramFile, &bigramFile);

   printf(" - DataStat ver %s\n", VERSION);
   printf(" | Pattern file (inputs, targets) [%s]\n", argv[PATTERNFILE]);
   printf(" +-> Output files [%s", argv[OUTPUTBASE]);

   unigramFile == stdout ? printf(" (stdout)]\n") : printf("].unigram, .bigram\n");

   ReadTags(tagsFile, &tagsArray, &numOutputs);
   printf("Read %d tags (%d outputs)\n", numOutputs, numOutputs);

   DoUnigramStats(patternFile, unigramFile, numOutputs, tagsArray);

   rewind(patternFile);
   DoBigramStats(patternFile, bigramFile, numOutputs, tagsArray);

   CloseFiles(tagsFile, patternFile, unigramFile, bigramFile);

   return 0;
}
```

```
/* --- Helper functions --- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** DataStat ver %s -- Calculates dataset statistics\n", VERSION);
   fprintf(output, "Usage: %s {tagsFile} {patternFile} {outputBase}\n", appName);
   fprintf(output, "Where: {tagsFile}    -- tags representing input lines\n");
   fprintf(output, "       {patternFile} -- file to specify output targets\n");
   fprintf(output, "       {outputBase}  -- base name of files to write output to\n");
   fprintf(output, "                        writes to {base}.unigram, {base}.bigram\n");
   fprintf(output, "                        (specify '-' for stdout)\n");
   fprintf(output, "For detailed info, type %s --help\n\n", appName);
}

void ProcessCommandLine(int argc, char *argv[], FILE **tagsFile, FILE **patternFile, FILE
**unigramFile, FILE **bigramFile)
{
   char  buffer[MAX_STRING];

   if ((!(argc < 2)) && ((strcasecmp(argv[1], "--help") == 0))) {
      Help(argv[APPNAME]);
      exit(0);
   }

   if (argc < NUMARGS) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      Abort(1);
   }

   if (argc > NUMARGS)
      fprintf(stderr, "--- Warning: extra arguments ignored.\n");

   if ((*tagsFile = fopen(argv[TAGSFILE], "rt")) == NULL) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Could not open tags file [%s].\n", argv[TAGSFILE]);
      Abort(3);
   }


   if ((*patternFile = fopen(argv[PATTERNFILE], "rt")) == NULL) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Could not open pattern file [%s].\n", argv[PATTERNFILE]);
      Abort(5);
   }

   if (strcasecmp(argv[OUTPUTBASE], "-") == 0)
      *unigramFile = *bigramFile = stdout;
   else {
      strcpy(buffer, argv[OUTPUTBASE]);
      strcat(buffer, ".unigram");

      if ((*unigramFile = fopen(buffer, "wt")) == NULL) {
         Usage(stderr, argv[APPNAME]);
         fprintf(stderr, "*** Could not open unigram output file [%s].\n", buffer);
      }

      strcpy(buffer, argv[OUTPUTBASE]);
      strcat(buffer, ".bigram");

      if ((*bigramFile = fopen(buffer, "wt")) == NULL) {
         Usage(stderr, argv[APPNAME]);
         fprintf(stderr, "*** Could not open bigram output file [%s].\n", buffer);
      }
   }
}

void Abort(int retnum)
{
   fprintf(stderr, "Aborting...\n");
   exit(retnum);
}

void Help(char *appName)
{
   FILE *helpFile;

   if ((helpFile = fopen(HELPFILE, "wt")) == NULL) {
      fprintf(stderr, "*** Could not create help file [%s].\n", HELPFILE);
      Abort(2);
   }
```

```
   fprintf(helpFile, "----------------------------------------\n");
   fprintf(helpFile, "   Help file for DataStat ver %s\n", VERSION);
   fprintf(helpFile, "----------------------------------------\n\n");

   Usage(helpFile, appName);

   fprintf(helpFile, "DatasetStats will calculate and output the unigram and bigram statistics\n");
   fprintf(helpFile, "for for datasets represented by a .pattern file.\n\n");

   fprintf(helpFile, ".pattern file format:\n");
   fprintf(helpFile, "   (pattern num -- zero origin) (predecessor) (successor)\n");
   fprintf(helpFile, "   .\n");
   fprintf(helpFile, "   .\n");
   fprintf(helpFile, "   .\n");
   fprintf(helpFile, "   <EOF>\n\n");

   fprintf(helpFile, "DatasetStats will use a .tags file (tags separated by whitespace) to\n");
   fprintf(helpFile, "prettyprint the stats marked with the category names.\n\n");

   fprintf(helpFile, "DatasetStats is known to work on a 29-class problem successfully.\n");

   fprintf(stderr, "Created %s\n", HELPFILE);
}


void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs)
{
   tokenList   *allTags, *index;
   int      numTags;

   allTags = ReadWholeFile(tagsFile);

   if (allTags == NULL) {
      fprintf(stderr, "*** Error reading token file.\n");
      Abort(5);
   }

   numTags = 0;
   index = allTags;
   while (index != NULL) {
      index = index -> NEXT;
      numTags++;
   }

   *numOutputs = numTags;

   if (!Allocate2DArray(tagsArray, numTags, 1, MAXTAGLENGTH)) {
      fprintf(stderr, "*** Could not allocate tag array.\n");
      Abort(6);
   }

   index = allTags;
   numTags = 0;
   while (index != NULL) {
      strcpy((*tagsArray)[numTags], index -> token);
      index = index -> NEXT;
      numTags++;
   }
   DestroyTokenList(allTags);
}

tokenList *ReadWholeFile(FILE *input)
{
   tokenList   *list, *temp;
   bool     inFile;

   inFile = TRUE;
   list = ReadLineTokens(input, &inFile);    /* Get the first line */

   if (list == NULL)
      return list;

   while (inFile) {
      temp = ReadLineTokens(input, &inFile); /* Get the next line */
      list = ConcatenateTokenList(list, temp);
   }

   return list;
}


void CloseFiles(FILE *tagsFile, FILE *patternFile, FILE *unigramFile, FILE *bigramFile)
```

```
{
   fclose(tagsFile);
   fclose(patternFile);
   fclose(unigramFile);
   fclose(bigramFile);
}

void DoUnigramStats(FILE *patternFile, FILE *outputFile, int numOutputs, char *tagsArray[])
{
   long  *outputCounts, numPatterns;
   bool  inFile;
   int   predecessor, successor, outputIndex;

   if ((outputCounts = (long *) malloc(sizeof(long) * numOutputs)) == NULL) {
      fprintf(stderr, "*** Could not allocate unigram stats table.\n");
      Abort(3);
   }

   outputIndex = 0;
   while (outputIndex < numOutputs)
      outputCounts[outputIndex++] = 0;

   numPatterns = 0;
   inFile = TRUE;
   while (inFile) {
      ReadPattern(patternFile, &predecessor, &successor, &inFile);

      outputCounts[successor - 1]++;
      numPatterns++;
   }

   OutputUnigramStats(tagsArray,     outputCounts,     outputFile,     numOutputs,     numPatterns,
outputCounts[MaxLongArray(outputCounts, numOutputs)]);
}


void ReadPattern(FILE *patternFile, int *predecessor, int *successor, bool *inFile)
{
   int   patternNum;

   fscanf(patternFile, "%d %d %d", &patternNum, predecessor, successor);

   if (feof(patternFile))
      *inFile = FALSE;
}


void OutputUnigramStats(char *tagsArray[], long outputCounts[], FILE *outputFile, int numOutputs,
int numPatterns, double unigramScore){
   int   category;
   long  countSum;

   fprintf(outputFile, "---- Unigram stats\n");
   fprintf(outputFile, " %d categories\n", numOutputs);
   fprintf(outputFile, " Unigram score: %4.2f%%\n", unigramScore / (double) numPatterns * 100.0);
   fprintf(outputFile, "(category) (count) (percentage)\n");

   category = 0;
   while (category < numOutputs) {
      fprintf(outputFile,  "%9s    %6d    %10.2f%%\n",  tagsArray[category],  outputCounts[category],
(float) outputCounts[category] / (float) numPatterns * 100.0);
      category++;
   }

   fprintf(outputFile, "-----------------------------\n");
   fprintf(outputFile, "           %6d  %10.2f%%\n", numPatterns, 100.0);

}

void DoBigramStats(FILE *patternFile, FILE *outputFile, int numOutputs, char *tagsArray[])
{
   long  **bigramCounts, bigramScore;
   int   predecessor, successor, *transitions,
      numPatterns;
   bool  inFile;

   if (!Allocate2DArray((char ***) &bigramCounts, numOutputs, numOutputs, sizeof(long))) {
      fprintf(stderr, "*** Could not allocate bigram stats array.\n");
      Abort(4);
   }

   for (predecessor = 0; predecessor < numOutputs; predecessor++)
```

```
      for (successor = 0; successor < numOutputs; successor++)
         bigramCounts[predecessor][successor] = 0;

   numPatterns = 0;
   inFile = TRUE;
   while (inFile) {
      ReadPattern(patternFile, &predecessor, &successor, &inFile);

      bigramCounts[predecessor - 1][successor - 1]++;
      numPatterns++;
   }

   if ((transitions = (int *) malloc(sizeof(int) * numOutputs)) == NULL) {
      fprintf(stderr, "*** Could not allocate trainsitions array.\n");
      Abort(5);
   }

   bigramScore = 0;
   for (predecessor = 0; predecessor < numOutputs; predecessor++) {
      transitions[predecessor] = MaxLongArray(bigramCounts[predecessor], numOutputs);
      bigramScore += bigramCounts[predecessor][transitions[predecessor]];
   }

   OutputBigramStats(tagsArray, bigramCounts, transitions, outputFile, numOutputs, numPatterns,
bigramScore);
}

void OutputBigramStats(char *tagsArray[], long **bigramCounts, int *transitions, FILE *outputFile,
int numOutputs, int numPatterns, double bigramScore)
{
   int   predecessor, successor;

   fprintf(outputFile, "---- Bigram stats\n");
   fprintf(outputFile, " %d categories\n", numOutputs);
   fprintf(outputFile, " Bigram score: %4.2f%%\n", bigramScore / (double) numPatterns * 100.0);

   fprintf(outputFile, " - - - - - - - - - - - - - - - - - - - - - - - - -\n");

   /* Counts */

   fprintf(outputFile, "            ");
   for (predecessor = 0; predecessor < numOutputs; predecessor++)
      fprintf(outputFile, "%9s   ", tagsArray[predecessor]);

   fprintf(outputFile, "\n");

   for (predecessor = 0; predecessor < numOutputs; predecessor++) {
      fprintf(outputFile, "%9s   ", tagsArray[predecessor]);

      for (successor = 0; successor < numOutputs; successor++)
         fprintf(outputFile, "%9d   ", bigramCounts[predecessor][successor]);

      fprintf(outputFile, "\n");
   }

   fprintf(outputFile, "------------------------------\n");
   fprintf(outputFile, "   %d patterns total\n\n\n", numPatterns);

   /* Percentages */

   fprintf(outputFile, "            ");
   for (predecessor = 0; predecessor < numOutputs; predecessor++)
      fprintf(outputFile, "%9s   ", tagsArray[predecessor]);

   fprintf(outputFile, "\n");

   for (predecessor = 0; predecessor < numOutputs; predecessor++) {
      fprintf(outputFile, "%9s   ", tagsArray[predecessor]);

      for (successor = 0; successor < numOutputs; successor++)
         fprintf(outputFile, "%8.2f%%    ", (float) bigramCounts[predecessor][successor] / (float)
numPatterns * (float) 100);

      fprintf(outputFile, "\n");
   }

   fprintf(outputFile, "------------------------------\n");
   fprintf(outputFile, "   100%% total\n\n\n");

   fprintf(outputFile, "\n\n");
   fprintf(outputFile, "Most common transitions\n");
   fprintf(outputFile, "------------------------------\n");
```

```
   for (predecessor = 0; predecessor < numOutputs; predecessor++)
      fprintf(outputFile,        "%9s                         %9s\n",        tagsArray[predecessor],
tagsArray[transitions[predecessor]]);

   fprintf(outputFile, "------------------------------\n");
}

bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize)
{
   int   index;

   if ((*array = malloc(xSize * sizeof(void *))) == NULL)
      return FALSE;

   for (index = 0; index < xSize; index++)
      if (((*array)[index] = malloc(ySize * elementSize)) == NULL) {
         free (*array);
         return FALSE;
      }

   return TRUE;
}

int MaxLongArray(long array[], int numElements)
{
   long  max = 0.0;
   int   mindex = 0;

   while (numElements > 0) {
      if (array[numElements - 1] > max) {
         max = array[numElements - 1];
         mindex = numElements - 1;
      }

      numElements--;
   }

   return mindex;
}


/* --- END of dstat.c --- */
```

9.2.4 SymStrip

Application purpose

SymStrip takes a transcribed language corpus with the words tagged with their word type (vowel, noun, etc) and separates the tags and words into separate text files. Symstrip can mark superfluous tags and insert reset markers at sentence boundaries.

Usage

2 input files are needed:

- A tags file, with whitespace-separated tokens that are the tags of interest. All other tokens will be "stripped" into the "words" file.
- A marked-up corpus, in a text file. SymStrip does not know about tag definitions like enclosing braces. The entire tag, verbatim, must be placed into the "tags" file. Tags and words must be whitespace-separated in the corpus. The code could be modified to incorporate this type of tagging, but it would probably be easier to run the corpus through a script to insert spaces.

SymStrip will process the corpus to extract the defined tags into one output file, and the other text within the corpus into another output file.

Modules used

TokenLst, TokScan, StdDefs, HTable

Source code

Makefile

```
symstrip: ui.o symsyms.o symstrip.o symcmds.o htable.o symopts.o tokenlst.o

symsyms.o: symsyms.c

symcmds.o: symcmds.c

htable.o: htable.c

ui.o: ui.c

symopts.o: symopts.c

tokenlst.o: tokenlst.c

symstrip.o: symstrip.c
```

ui.h

```
/* ui.h -- User interface for symbolstripper
 */

#ifndef  __UI_H
#define  __UI_H

#include "tokenlst.h"
#include "stddefs.h"


/* --- Lexical enumeration --- */

enum CommandsTag {exec_symb,
                     exec_symb_init,
                     exec_symb_read,
                    exec_symb_add,
                     exec_symb_del,
                     exec_symb_clear,
                     exec_symb_linfo,
                    exec_symb_help,

                 exec_opt,
                    exec_opt_disp,
```

```
                     exec_opt_load,
                     exec_opt_save,
                     exec_opt_def,
                     exec_opt_sfile,
                     exec_opt_hsize,
                     exec_opt_extra,
                     exec_opt_extratag,
                     exec_opt_eos,
                     exec_opt_eostag,
                     exec_opt_help,

                  exec_end, exec_strip, exec_core, exec_help, exec_info, exec_exit, INVALID};

typedef  enum CommandsTag  command;

enum MenusTag {menu_main, menu_symb, menu_opt};

typedef  enum MenusTag     menus;


/* -- Function prototypes -- */

void Help(void);
void Header(void);
void Footer(void);
void Info(void);
void Init(void);
void Cleanup(void);
void Prompt(void);
void CoreCheck(unsigned begin, unsigned end);
void SyntaxError(tokenList *comm);
void ExecuteMain(command comm);
bool StringCompI(char *string1, char *string2);
tokenList *GetInput(string buffer, int length);

#endif   /* __UI_H */

/* --- END of ui.h --- */
```

ui.c

```
/* UI -- User Interface for Symbol Stripper
 *
 * Author: Dylan Muir
 * Date: 11th Feburary 1999
 * Project: Language Processing Group -- QUT MLRC -- Sem 1 1999
 */

#include <stdlib.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "ui.h"
#include "stddefs.h"
#include "tokenlst.h"
#include "htable.h"
#include "symcmds.h"


/* --- Function prototypes --- */

command SyntaxCheck(tokenList *comm);
command SyntaxCheckMain(tokenList *comm);
void ExecuteCommand(command comm);


/* --- Globals --- */

time_t   tm;                  /* timestamp of program execution */
symOpts  options;             /* variables (defined in "symcmds.h") */
menus    menu;                /* Current execution menu */


/* --- Main --- */

int main(void)
{
   char        line[MAXSTRING];
```

```
   tokenList   *commands,
               *index;
   bool        contExec;            /* Should we continue execution? */
   command     comm;                /* Currently read command */

   unsigned    begin, end;          /* variables for checking memory usage */

   Header();                        /* Display program header */

   Init();                          /* Initiallise system */

   Help();                          /* Show help info */

#ifdef   DEBUG
/*   begin = coreleft(); */
#endif /* DEBUG */

   contExec = TRUE;

   while (contExec) {
      Prompt();               /* Display prompt */
      index = commands = GetInput(line, MAXSTRING);

      while (index != NULL) {
         if ((comm = SyntaxCheck(index)) == INVALID)
            SyntaxError(index);

         else if (comm == exec_exit) {
            contExec = FALSE;
            break;

         } else
            ExecuteCommand(comm);

         index = index -> NEXT;
      }

      commands = DestroyTokenList(commands);

#ifdef   DEBUG
/*      end = coreleft(); */

      CoreCheck(begin, end);

/*       begin = coreleft(); */
#endif /* DEBUG */
   }

   Footer();

   return 0;
}


/* --- Function bodies --- */

/* -- Function Help
 * Pre: TRUE
 * Post: Help info was printed
 */
void Help(void)
{
   printf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n\n",
          "Commands:",
          "| strip - strip symbols",
          "| symbols - symbols commands",
          "| options - options commands",
          "| core - display the amout of free core memory",
          "| help - this screen",
          "| info - display info about the test driver",
          "| exit - leave program",
          "|",
          "| * -- Not implemented!");
}


/* -- Function Header
 * Pre: TRUE
 * Post: Test driver header info was printed
 */
void Header(void)
{
```

```
   tm = time(NULL);

   printf("\n%s\n%s%s%s\n%s\n%s\n%s %s%s\n\n",
          "+------------------------------+",
          "| SymbolStripper ver ", VERSION, "      |",
          "| QUT MLRC LPG (Sem 1 1999)     |",
          "+------------------------------+",
          "Executed on", ctime(&tm),
          "--------------------------------------------------------");
}


/* -- Function Footer
 * Pre: TRUE
 * Post: Test driver footer info was printed
 */
void Footer(void)
{
   tm = time(NULL);

   printf("%s\n%s%s\n%s %s\n",
          "-------------------------------------------------------",
          "QUT MLRC LPG -- SymbolStripper ver ", VERSION,
          "Terminated on", ctime(&tm));
}


/* -- Function Info
 * Pre: TRUE
 * Post: Test driver info was printed
 */
void Info(void)
{
   printf("\n%s %s\n%s\n%s\n%s %s %s\n%s %s%s %d\n%s %d\n\n",
          "SymbolStripper ver", VERSION,
          " Author: Dylan Muir [dr.muir@student.qut.edu.au]",
          " Project: QUT MLRC LPG (Sem1 1999)",
          " Compile time:", __TIME__, __DATE__,
          " Execution time:", ctime(&tm),
          " Max string:", MAXSTRING,
          " Max lookup key size:", MAX_KEY_SIZE);
}


/* -- Function Init
 * Pre: TRUE
 * Post: Performs any initiallisation required
 */
void Init(void)
{
   menu = menu_main;          /* Set main menu */

   InitOptionsCommand();      /* \__ Defined in SymCmds.h */
   LoadOptionsCommand();      /* /                        */
}

/* -- Function Cleanup
 * Pre: TRUE
 * Post: Cleans up anything necessary
 */
void Cleanup(void)
{
   if (options.symbolsRead)
      ClearSymbolsCommand();     /* Clear symbols if necessary */

   if (options.hashInit)
      hashDestroyTable();        /* Destroy lookup table if necessary */
}


/* -- Function Prompt
 * Pre: TRUE
 * Post: the prompt for 'menu' was output
 */
void Prompt(void)
{
   switch (menu) {
      case menu_main:
         printf("] ");
         break;

      case menu_symb:
```

```
         printf("symbols> ");
         break;

      case menu_opt:
         printf("options> ");
         break;

      default:
         printf("> ");
   }
}


/* -- Function CoreCheck
 * Pre:
 * Post:
 */
void CoreCheck(unsigned begin, unsigned end)
{
   if (begin != end)
      printf("Core discrepancy: %+d\n", ((int) end) - ((int) begin));
}


/* -- Function Getinput
 * Pre: 'buffer' has been allocated
 * Post: a line of text was read from stdin, converted to a list of tokens and returned
 */
tokenList *GetInput(string buffer, int length)
{
   char        read,             /* Character read from stdin */
               *index;           /* index into buffer */

   index = buffer;         /* Reset index */

   while (length > 0) {
      length--;
      if ((read = (char) getchar()) == EOL)
         length = 0;
      else
         *index++ = read;
   }
   *index = EOS;        /* Nul-terminate */

   printf("\n");

   return GetTokens(buffer);
}


/* -- Function SyntaxCheck
 * Pre:
 * Post
 */
command SyntaxCheck(tokenList *comm)
{
   switch (menu) {
      case menu_main:
         return SyntaxCheckMain(comm);

      case menu_symb:
         return SyntaxCheckSymbol(comm);

      case menu_opt:
         return SyntaxCheckOption(comm);

      default:
         return exec_end;
   }
}


/* -- Function SyntaxCheckMain
 * Pre:
 * Post
 */
command SyntaxCheckMain(tokenList *comm)
{

   /* Un-implemented commands */

   /*--- beginning of unimplemented commands  ---*-
```

```
   -*--- end of unimplemented commands        ---*/

   if (StringCompI(comm -> token, "strip"))
      return exec_strip;

   if (StringCompI(comm -> token, "symbols"))
      return exec_symb;

   if (StringCompI(comm -> token, "options"))
      return exec_opt;

   if (StringCompI(comm -> token, "core"))
      return exec_core;

   if (StringCompI(comm -> token, "help"))
      return exec_help;

   if (StringCompI(comm -> token, "info"))
      return exec_info;

   if (StringCompI(comm -> token, "exit"))
      return exec_exit;

   return INVALID;
}



/* -- Function SyntaxError
 * Pre:
 * Post:
 */
void SyntaxError(tokenList *comm)
{
   printf("Command not recognised: %s.\n\n", comm -> token);
}


/* -- Function ExecuteCommand
 * Pre:
 * Post:
 */
void ExecuteCommand(command comm)
{
   if (comm == exec_end) {
      menu = menu_main;
      return;
   }

   switch (menu) {
      case menu_symb:
         ExecuteSymbol(comm);
         return;

      case menu_opt:
         ExecuteOption(comm);
         return;

      case menu_main:
      default:
         ExecuteMain(comm);
         return;
   }
}


/* -- Function ExecuteMain
 * Pre:
 * Post:
 */
void ExecuteMain(command comm)
{
   switch (comm) {
      case exec_strip:
         StripCommand();
         break;

      case exec_symb:
         menu = menu_symb;
         SymbolHelpCommand();
         break;
```

```
      case exec_opt:
         menu = menu_opt;
         OptionHelpCommand();
         break;

      case exec_core:
        printf("Not implemented on Unix\n\n");
/*       printf("Free memory: %u\n\n", coreleft()); */
         break;

      case exec_info:
         Info();
         break;

      case exec_help:
         Help();
         break;

      default:
         ;/* Should never be reached */
   }
}


/* -- Function StringCompI
 * Pre: 'string1' and 'string2' are valid nul-terminated strings
 * Post: ('string' == 'string2' && TRUE was returned) ||
 *       ('string1' != 'string2' && FALSE was returned)
 */
bool StringCompI(char *string1, char *string2)
{
   char  *copy1 = string1,
         *copy2 = string2;

   if (strlen(string1) != strlen(string2))
      return FALSE;

   {  while (*copy1 != '\0') {
         if (toupper(*copy1++) != toupper(*copy2++))
            return FALSE;
      }
   }
   return TRUE;
}
```

symcmds.h

```
/* symcmds.h -- include file for symbolstripper ui
 *
 * Author: Dylan Muir
 * Date: 12th February, 1999
 * Modified: 15th Fenruary, 1999
 * Version: 1.04
 *
 * Purpose: Holds definitions for non-ui commands and options struct definition
 */

#ifndef  __SYMCMDS_H
#define  __SYMCMDS_H

#include "stddefs.h"
#include "ui.h"
#include "tokenlst.h"


/* -- SymbolStripper definitions -- */

#define  OPTIONS_FILE   "symopts.opt"
#define  VERSION        "1.05"

/* -- Options structure -- */

struct symOptsTag {
   char  symbolFile[MAXSTRING];            /* default file for symbols */
   bool  symbolsRead;                      /* have symbols been read in? */

   unsigned hashSize;                      /* size of hash table */
   bool     hashInit;                      /* has the htable been initiallised? */

   bool  markExtra;                        /* mark extra tags per word? */
   char  markExtraTag[MAXSTRING];          /* symbol to mark extra tags with */
```

```
   bool  markEOS;                           /* mark end of sentances? */
   char  markEOSTag[MAXSTRING];             /* symbol to mark eos with */
};

typedef struct symOptsTag symOpts;


/* -- Command functions -- */

void     StripCommand(void);

void     SymbolHelpCommand(void);
command  SyntaxCheckSymbol(tokenList *comm);
void     ExecuteSymbol(command comm);

void     InitSymbolsCommand(void);
void     ReadSymbolsCommand(void);
void     AddSymbolsCommand(void);
void     DeleteSymbolsCommand(void);
void     ClearSymbolsCommand(void);
void     LookupInfoSymbolsCommand(void);


void     OptionHelpCommand(void);
command  SyntaxCheckOption(tokenList *comm);
void     ExecuteOption(command comm);

void     InitOptionsCommand(void);
void     LoadOptionsCommand(void);
void     SaveOptionsCommand(void);
void     LoadDefaultOptionsCommand(void);
void     DisplayOptionsCommand(void);
void     HashSizeOptionsCommand(void);
void     SymbolFileOptionsCommand(void);
void     ExtraOptionsCommand(void);
void     ExtraTagOptionsCommand(void);
void     EOSOptionsCommand(void);
void     EOSTagOptionsCommand(void);


/* -- Helper functions -- */

/* -- Function GetInputFileName
 * Pre: 'buffer' has been allocated and prompt has been printed
 * Post: 'buffer' contains a string that could be opened for reading
 */
void GetInputFileName(char *fName);

/* -- Function GetNewFileName
 * Pre: 'buffer' has been allocated and prompt has been printed
 * Post: 'buffer' contains a string that could be created as a file
 */
void GetNewFileName(char *fName);

/* -- Function ReadInput
 * Pre: 'buffer' has been allocated, and 'length' <= len(buffer)
 * Post: Up to 'length' characters have been read from stdin, and are in 'buffer'
 */
void ReadInput(char *buffer, unsigned length);

#endif   /* __SYMCMDS_H */

/* --- END of symcmds.h --- */
```

SymCmds.c

```
/* SymCmds.c -- functions called by UI
 *
 * See SymCmds.h for details
 */

#include <stdio.h>
#include "symcmds.h"
#include "symstrip.h"


void StripCommand(void)
{
   char  inputFile[MAXSTRING],        /* file to process */
         tagsFile[MAXSTRING],         /* file to strip tags to */
```

```
         wordsFile[MAXSTRING];           /* file to strip words to */

   printf("Enter an input file\n");
   GetInputFileName(inputFile);

   printf("\nEnter a file to strip tags to\n");
   GetNewFileName(tagsFile);

   printf("\nEnter a file to strip words to\n");
   GetNewFileName(wordsFile);

   printf("\nStripping file [%s]\n", inputFile);
   printf(" |\n");
   printf(" +--(tags)--> [%s]\n", tagsFile);
   printf(" |\n");
   printf(" +--(words)-> [%s]\n", wordsFile);
   printf(" \n");

   SymStrip(inputFile, tagsFile, wordsFile);
}


void GetInputFileName(char *fName)
{
   FILE  *fTest;             /* Used for testing validity of filename */
   bool  validFile;
   char  buffer[MAXSTRING];

   validFile = FALSE;

   while (!validFile) {
      printf("file> ");
      ReadInput(buffer, MAXSTRING);
      sscanf(buffer, "%s", fName);

      if ((fTest = fopen(fName, "rb")) != NULL) {
         fclose(fTest);
         validFile = TRUE;
         break;
      }

      printf("Unable to open [%s]\n", fName);
   }
}

void GetNewFileName(char *fName)
{
   FILE  *fTest;             /* Used for testing validity of filename */
   bool  validFile;
   char  buffer[MAXSTRING];

   validFile = FALSE;

   while (!validFile) {
      printf("file> ");
      ReadInput(buffer, MAXSTRING);
      sscanf(buffer, "%s", fName);

      if ((fTest = fopen(fName, "wb+")) != NULL) {
         fclose(fTest);
         validFile = TRUE;
         break;
      }

      printf("Unable to create [%s]\n", fName);
   }
}

void ReadInput(char *buffer, unsigned length)
{
   char  read;

   while (length > 0) {
      length--;
      if ((read = (char) getchar()) == EOL)
         length = 0;
      else
         *buffer++ = read;
   }
   *buffer = EOS;      /* Nul-terminate */
}
```

```
/* --- END or symcmds.c --- */
```

SymSyms.c

```
/* Symsyms.c -- commands file for symbolstripper UI
 *
 * See SymCmds.h for details
 */

#include <stdio.h>
#include "htable.h"
#include "symcmds.h"
#include "symstrip.h"

extern symOpts options;


void SymbolHelpCommand(void)
{
   printf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n\n",
          "Symbols commands:",
          "| init - (re)initiallise lookup system",
          "| read - read in symbols from file",
          "| add - add symbols",
          "| delete - delete symbols",
          "| clear - clear all symbols and destroy lookup table",
          "| linfo - display info about lookup table",
          "| help - this page",
          "| end - return to main command input");
}


command SyntaxCheckSymbol(tokenList *comm)
{
   /* Un-implemented commands */

   /*--- beginning of unimplemented commands  ---*-
   -*--- end of unimplemented commands        ---*/

   if (StringCompI(comm -> token, "init"))
      return exec_symb_init;

   if (StringCompI(comm -> token, "read"))
      return exec_symb_read;

   if (StringCompI(comm -> token, "add"))
      return exec_symb_add;

   if (StringCompI(comm -> token, "delete"))
      return exec_symb_del;

   if (StringCompI(comm -> token, "clear"))
      return exec_symb_clear;

   if (StringCompI(comm -> token, "linfo"))
      return exec_symb_linfo;

   if (StringCompI(comm -> token, "help"))
      return exec_symb_help;

   if (StringCompI(comm -> token, "end"))
      return exec_end;


   return INVALID;
}


void  ExecuteSymbol(command comm)
{
   switch (comm) {
      case exec_symb_init:
         InitSymbolsCommand();
         break;

      case exec_symb_read:
         ReadSymbolsCommand();
         break;

      case exec_symb_add:
         AddSymbolsCommand();
```

```
         break;

      case exec_symb_del:
         DeleteSymbolsCommand();
         break;

      case exec_symb_clear:
         ClearSymbolsCommand();
         break;

      case exec_symb_linfo:
         LookupInfoSymbolsCommand();
         break;

      case exec_symb_help:
         SymbolHelpCommand();
         break;

      default:
         ;/* Should never be reached */
   }
}


void InitSymbolsCommand(void)
{
   if (options.hashInit)
      hashDestroyTable();

   options.hashInit = hashInitTable(options.hashSize);
}


void ReadSymbolsCommand(void)
{
   FILE           *symbolFile;                   /* file pointer */
   bool           inFile;                        /* are we in the file? */
   tokenList      *tokens,                       /* list of tokens extracted from file */
                  *index;                        /* index into token list */
   char           *text;                         /* temp token text holder */
   long unsigned  symCount;                      /* number of symbols loaded */

   if (options.symbolsRead)
      ClearSymbolsCommand();                /* Clear tokens if already read */

   if (!options.hashInit)                   /* ensure table is initiallised */
      options.hashInit = hashInitTable(options.hashSize);

   printf("Reading symbols from [%s]...\n", options.symbolFile);

   symbolFile = fopen(options.symbolFile, "rt");  /* open file */

   if (symbolFile == NULL) {                /* error opening file */
      printf("Could not open [%s]\n", options.symbolFile);
      printf("*** SYMBOLS NOT READ\n");
      return;
   }

   inFile = TRUE;                           /* we are in the file */
   symCount = 0;

   while (inFile) {
      tokens = ReadLineTokens(symbolFile, &inFile);

      index = tokens;

      while (index != NULL) {
         text = index -> token;             /* Avoid unnecessary indexing */

         if (!hashIsValidKey(text))
            printf("*** Symbol [%s] not a valid key\n", text);

         else if (hashIsIn(text))
            printf("*** Symbol [%s] already in table\n", text);

         else if (hashIsFull())
            printf("*** Symbol [%s] not added -- table full\n", text);

         else {
            if (!hashInsert(index -> token))
               printf("*** Symbol [%s] not added -- undefined error\n", text);
```

```
            else {
               symCount++;
               if (hashLoadFactor() > 80)
                  printf("--- Warning: lookup table load factor is %u\%\n", hashLoadFactor());
            }

         }

         index = index -> NEXT;
      }

      DestroyTokenList(tokens);
   }

   fclose(symbolFile);

   options.symbolsRead = TRUE;

   printf("...loaded %u symbols\n\n", symCount);
}


void AddSymbolsCommand(void)
{
   tokenList   *symbols,            /* list of symbols to add */
               *index;              /* index into otoken list */
   char        buffer[MAXSTRING],   /* buffer for input */
               *text;               /* temp to prevent excessive indexing */

   if (!options.hashInit) {
      hashInitTable(options.hashSize);
      options.hashInit = TRUE;
   }

   printf("Enter symbols to add, separated by a space\n");

   printf("add> ");
   index = symbols = GetInput(buffer, MAXSTRING);

   while (index != NULL) {
      text = index -> token;              /* Avoid unnecessary indexing */

      if (!hashIsValidKey(text))
         printf("*** Symbol [%s] not a valid key\n", text);

      else if (hashIsIn(text))
         printf("*** Symbol [%s] already in table\n", text);

      else if (hashIsFull())
         printf("*** Symbol [%s] not added -- table full\n", text);

      else
         if (!hashInsert(index -> token))
            printf("*** Symbol [%s] not added -- undefined error\n", text);

      if (hashLoadFactor() > 80)
         printf("--- Warning: lookup table load factor is %d\%\n");

      index = index -> NEXT;
   }

   DestroyTokenList(symbols);

   printf("...done\n\n");
}


void DeleteSymbolsCommand(void)
{
   tokenList   *symbols,            /* list of symbols to delete */
               *index;              /* index into otoken list */
   char        buffer[MAXSTRING],   /* buffer for input */
               *text;               /* temp to prevent excessive indexing */

   if (!options.hashInit) {
      printf("No symbols to delete\n");
      return;
   }

   printf("Enter symbols to delete, separated by a space\n");

   printf("del> ");
```

```
   index = symbols = GetInput(buffer, MAXSTRING);

   while (index != NULL) {
      text = index -> token;             /* Avoid unnecessary indexing */

      if (!hashIsValidKey(text))
         printf("*** Symbol [%s] not a valid key\n", text);

      else if (hashIsEmpty())
         printf("*** Symbol [%s] not deleted -- table empty\n", text);

      else if (!hashIsIn(text))
         printf("*** Symbol [%s] is not in table\n", text);

      else
         if (!hashDelete(index -> token))
            printf("*** Symbol [%s] not deletd -- undefined error\n", text);

      index = index -> NEXT;
   }

   DestroyTokenList(symbols);

   printf("...done\n\n");
}

void ClearSymbolsCommand(void)
{
   if (!options.hashInit)                /* No symbols to clear */
      return;

   printf("Clearing all symbols...");

   if (options.hashInit) hashDestroyTable(); /* Destroy table */
   options.hashInit = FALSE;                 /* Set flags */
   options.symbolsRead = FALSE;

   printf(" done\n\n");
}

void LookupInfoSymbolsCommand(void)
{
   printf("Lookup table info:\n");
   if (!options.hashInit) {
      printf(" Table not initiallised\n\n");
      return;
   }

   printf(" Maximum entries: %u\n", options.hashSize);
   printf(" Loaded symbols: %u\n", hashNumEntries());
   printf(" Load factor: %u\%\n", hashLoadFactor());

   printf(" Table full? ");
   if (hashIsFull())
      printf("YES\n");
   else
      printf("NO\n");

   printf(" Table empty? ");
   if (hashIsEmpty())
      printf("YES\n");
   else
      printf("NO\n");

   printf(" Max key length: %d\n", MAX_KEY_SIZE);

   printf("\n");
}

/* --- END of symsyms.c --- */
```

SymOpts.c

```
/* Symopts.c -- manipulates system options
 *
 * See SymBmds.h for details
 */

#include <stdio.h>
#include <string.h>
#include "symcmds.h"
```

```
#include "htable.h"

extern symOpts options;


void OptionHelpCommand(void)
{
   printf("%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n%s\n\n",
          "Options commands:",
          "| display - show all options",
          "| load - load options",
          "| save - save options",
          "| default - load default options",
          "| symfile - specify symbol file",
          "| hashsize - specify hash table size",
          "| extra - toggle marking extra symbols per word",
          "| extratag - specify extra symbol mark tag",
          "| eos - toggle use of 'end of sentance' tag",
          "| eostag - specify 'end of sentance' tag",
          "| help - this page",
          "| end - return to main command input");
}


command SyntaxCheckOption(tokenList *comm)
{
   /* Un-implemented commands */

   /*--- beginning of unimplemented commands  ---*-
   -*--- end of unimplemented commands        ---*/

   if (StringCompI(comm -> token, "display"))
      return exec_opt_disp;

   if (StringCompI(comm -> token, "load"))
      return exec_opt_load;

   if (StringCompI(comm -> token, "save"))
      return exec_opt_save;

   if (StringCompI(comm -> token, "default"))
      return exec_opt_def;

   if (StringCompI(comm -> token, "symfile"))
      return exec_opt_sfile;

   if (StringCompI(comm -> token, "hashsize"))
      return exec_opt_hsize;

   if (StringCompI(comm -> token, "extra"))
      return exec_opt_extra;

   if (StringCompI(comm -> token, "extratag"))
      return exec_opt_extratag;

   if (StringCompI(comm -> token, "eos"))
      return exec_opt_eos;

   if (StringCompI(comm -> token, "eostag"))
      return exec_opt_eostag;

   if (StringCompI(comm -> token, "help"))
      return exec_opt_help;

   if (StringCompI(comm -> token, "end"))
      return exec_end;


   return INVALID;
}

void ExecuteOption(command comm)
{
   switch (comm) {
      case exec_opt_disp:
         DisplayOptionsCommand();
         break;

      case exec_opt_load:
         LoadOptionsCommand();
         break;
```

```
      case exec_opt_save:
         SaveOptionsCommand();
         break;

      case exec_opt_def:
         LoadDefaultOptionsCommand();
         break;

      case exec_opt_sfile:
         SymbolFileOptionsCommand();
         break;

      case exec_opt_hsize:
         HashSizeOptionsCommand();
         break;

      case exec_opt_extra:
         ExtraOptionsCommand();
         break;

      case exec_opt_extratag:
         ExtraTagOptionsCommand();
         break;

      case exec_opt_eos:
         EOSOptionsCommand();
         break;

      case exec_opt_eostag:
         EOSTagOptionsCommand();
         break;

      case exec_opt_help:
         OptionHelpCommand();
         break;

      default:
         ;/* Should never be reached */
   }
}


void InitOptionsCommand(void)
{
   strcpy(options.symbolFile, "");
   options.symbolsRead = FALSE;

   options.hashSize = 0;
   options.hashInit = FALSE;

   options.markExtra = FALSE;
   strcpy(options.markExtraTag, "");

   options.markEOS = FALSE;
   strcpy(options.markEOSTag, "");
}

void LoadOptionsCommand(void)
{
   FILE  *optionsFile;

   char  version[5];

   printf("Loading options from [%s]...\n", OPTIONS_FILE);

   optionsFile = fopen(OPTIONS_FILE, "rb");

   if (optionsFile == NULL) {
      printf("Unable to open [%s]\n", OPTIONS_FILE);
      printf("Loading default options...\n");

      LoadDefaultOptionsCommand();           /* Load defaults */
      DisplayOptionsCommand();               /* Show the defaults */
      SaveOptionsCommand();

   } else {
      fread(version, 5, 1, optionsFile);     /* first 5 bytes == version */

      if (strcmp(version, VERSION) != 0) {   /* compare SymbolStripper versions */
         printf("This options file was created with version [%s]\n", version);
         printf("You are running version [%s]\n", VERSION);
         printf("Loading default options...\n");
```

```
         LoadDefaultOptionsCommand();            /* Load defaults */
         DisplayOptionsCommand();                /* Show the defaults */

      } else {
         fread(&options, sizeof(options), 1, optionsFile); /* Read options */
         fclose(optionsFile);                        /* Close file */
      }
   }

   options.symbolsRead = FALSE;                  /* reset symbolsRead flag */
   options.hashInit = FALSE;                     /* reset hashInit flag */

   printf("...done.\n\n");
}


void SaveOptionsCommand(void)
{
   FILE  *optionsFile;

   char  version[5];

   printf("Saving options to [%s]...\n", OPTIONS_FILE);

   optionsFile = fopen(OPTIONS_FILE, "wb");

   if (optionsFile == NULL) {
      printf("Unable to open [%s]\n", OPTIONS_FILE);
      printf("*** OPTIONS NOT SAVED\n");

   } else {
      strcpy(version, VERSION);                  /* assign version string */

      fwrite(version, 5, 1, optionsFile);        /* version -> first five bytes */

      fwrite(&options, sizeof(options), 1, optionsFile);/* Write options */
      fclose(optionsFile);                       /* Close file */
   }

   printf("...done.\n\n");
}


void LoadDefaultOptionsCommand(void)
{
   if (options.hashInit)
      ClearSymbolsCommand();

   strcpy(options.symbolFile, "symbols.txt");
   options.symbolsRead = FALSE;

   options.hashSize = 201;
   options.hashInit = FALSE;

   options.markExtra = TRUE;
   strcpy(options.markExtraTag, "*");

   options.markEOS = TRUE;
   strcpy(options.markEOSTag, "/S");
}


void DisplayOptionsCommand(void)
{
   printf("\n--- SymbolStripper settings ---\n");

   printf("Current symbol file -- [%s]\n", options.symbolFile);

   printf("Symbols read? ");
   if (options.symbolsRead)
      printf("YES\n");
   else
      printf("NO\n");

   printf("Maximum lookup table size: %u\n", options.hashSize);

   printf("Lookup table initiallised? ");
   if (options.hashInit)
      printf("YES\n");
   else
      printf("NO\n");
```

```
   printf("Mark extra symbols per word with [%s]? ", options.markExtraTag);
   if (options.markExtra)
      printf("YES\n");
   else
      printf("NO\n");

   printf("Mark end of sentance with [%s]? ", options.markEOSTag);
   if (options.markEOS)
      printf("YES\n");
   else
      printf("NO\n");


   printf("\n");
}


void HashSizeOptionsCommand(void)
{
   unsigned hashSize;

   printf("Current maximum hash size: %u\n", options.hashSize);
   printf("Enter new maximum hash size: ");
   scanf("%u", &hashSize);

   if (hashSize == options.hashSize)         /* If it's the same, */
      return;                                /* don't do anything */

   else {
      if (options.hashInit)
         ClearSymbolsCommand();

      options.hashSize = hashSize;

      if (options.hashInit) {
         hashDestroyTable();
         options.hashInit = FALSE;
      }

      options.symbolsRead = FALSE;
   }

   printf("...done.\n");
}

void ExtraOptionsCommand(void)
{
   printf("Marking of extra symbols per word ");

   if (options.markExtra) {
      options.markExtra = FALSE;
      printf("OFF\n");

   } else {
      options.markExtra = TRUE;
      printf("ON\n");
   }
}

void ExtraTagOptionsCommand(void)
{
   printf("Current extra symbols tag: [%s]\n", options.markExtraTag);
   printf("Enter new extra symbols tag: ");
   scanf("%s", &options.markExtraTag);

   printf("...done\n");
}

void EOSOptionsCommand(void)
{
   printf("Marking of end of sentance is ");

   if (options.markEOS) {
      options.markEOS = FALSE;
      printf("OFF\n");

   } else {
      options.markEOS = TRUE;
      printf("ON\n");
   }
}
```

```
void EOSTagOptionsCommand(void)
{
   printf("Current eos tag: [%s]\n", options.markEOSTag);
   printf("Enter new eos tag: ");
   scanf("%s", &options.markEOSTag);

   printf("...done\n");
}


void SymbolFileOptionsCommand(void)
{
   char  symbolFile[MAXSTRING];       /* New symbol file name */

   printf("Current symbol file: [%s]\n", options.symbolFile);
   printf("Enter new symbol file\n");

   GetInputFileName(symbolFile);

   if (StringCompI(symbolFile, options.symbolFile))
      return;                              /* no change, so exit */

   strcpy(options.symbolFile, symbolFile);   /* assign new file */

   if (options.symbolsRead) {
      hashDestroyTable();                  /* Remove prior symbols */
      options.symbolsRead = FALSE;         /* Set flags */
      options.hashInit = FALSE;
   }

   printf("... done\n");
}
```

SymStrip.h

```
/* SymStrip.h -- Symbol stripping engine
 *
 * Author: Dylan Muir [dr.muir@student.qut.edu.au
 * Date: 14th February, 1999
 * Version: 1.00
 */

#ifndef  __SYMSTRIP_H
#define  __SYMSTRIP_H

#include <stdio.h>
#include "tokenlst.h"
#include "stddefs.h"


/* -- Function SymStrip
 * Pre: 'inputFile' is an existing file
 *      'tagsFile' and 'wordsFile' can be created
 * Post: 'inputFile' has been processed &&
 *       tags were stripped to 'tagsfile' &&
 *       words were strripped to 'wordsFile'
 */
void  SymStrip(string inputFile, string tagsFile, string wordsFile);


/* -- Function ReadLineTokens
 * Pre: 'file' is a text file opened for reading
 *      'inFile' has been allocated, and indicates wether EOF has been reached
 * Post: a line of text up to an EOL or EOF has been scanned and
 *       converted into a token list, which was returned
 */
tokenList   *ReadLineTokens(FILE *ifile, bool *inFile);

/* -- Function ReadLineText
 * Pre: 'file' is a text file opened for reading
 *      'buffer' has been allocated, and has max length 'length'
 *      'inFile' has been allocated, and indicated wether EOF has been reached
 * Post: 'buffer' contains a line of text up to EOL or EOF (defined in stddefs.h) &&
 *       'inFile' indicates wether EOF has been reached
 */
void  ReadLineText(FILE *ifile, char *buffer, unsigned length, bool *inFile);

#endif   /* __SYMSTRIP_H */

/* --- END of SymStrip.h --- */
```

SymStrip.c

```
/* symstrip.c -- Symbol stripping engine
 *
 * See symstrip.h for details
 */

#include "stddefs.h"
#include "symcmds.h"
#include "symstrip.h"
#include "tokenlst.h"
#include "htable.h"

extern symOpts options;

/* -- Local functions -- */

void NewLine(FILE *FP);
void MarkEOS(FILE *FP);
void MarkExtra(FILE *FP);
void WriteToken(FILE *FP, tokenList *token);
bool CheckSequence(FILE *tagsFP, FILE *wordsFP, string tag, string word, tokenList *index);

/* -- Exported functions -- */

void SymStrip(string inputFile, string tagsFile, string wordsFile)
{
   FILE  *inputFP,
         *tagsFP,
         *wordsFP;

   tokenList   *list,                  /* line of tokens from file */
               *index;                 /* index into token list */

   bool        inFile;                 /* Are we within the file? */
   bool        readTag;                /* Did we just read a tag? */
   unsigned    lineCount,              /* Number of lines */
               wordCount;              /* Number of words (non-tags) read */

   if (!options.symbolsRead) {                  /* Ensure we have some tags to work with! */
      printf("Reading symbols...\n");
      ReadSymbolsCommand();

      if (!options.symbolsRead) {               /* Unsuccessful symbol read */
         printf("Unable to open specified symbols file.\n");
         printf("Aborting...\n");
         return;                    /* So die */
      }
   }

   if ((inputFP = fopen(inputFile, "rt")) == NULL) {
      printf("Unable to open [%s] for input\n", inputFile);
      printf("Aborting...\n");
      return;
   }

   if ((tagsFP = fopen(tagsFile, "wt")) == NULL) {
      printf("Unable to open [%s] for tags output\n", tagsFile);
      printf("Aborting...\n");

      fclose(inputFP);
      return;
   }

   if ((wordsFP = fopen(wordsFile, "wt")) == NULL) {
      printf("Unable to open [%s] for words output\n", wordsFile);
      printf("Aborting...\n");

      fclose(inputFP);
      fclose(tagsFP);
      return;
   }


   inFile = TRUE;                               /* We are currently in the file */
   readTag = FALSE;                             /* No tags read yet */

   lineCount = 0;                               /* \___ Reset counts */
   wordCount = 0;                               /* /                 */
```

```
   printf("Stripping file...\n");

   while (inFile) {
      index = list = ReadLineTokens(inputFP, &inFile);   /* Get a line of tokens */

      while (index != NULL) {
         if (hashIsIn(index -> token)) {        /* Is it a tag? */
         if (readTag)                           /* Is it an extra tag? */
                 MarkExtra(tagsFP);                   /* Mark it */

      WriteToken(tagsFP, index);             /* Strip to tags file */
      readTag = TRUE;                     /* We just read a tag */

         } else {
            WriteToken(wordsFP, index);         /* Strip to words file */
            readTag = FALSE;                    /* We didn't read a tag */

            wordCount++;                        /* read another word */
         }

         index = index -> NEXT;                 /* Next tokoen */
      }

      DestroyTokenList(list);

      MarkEOS(tagsFP);                          /* End of sentance */
      NewLine(tagsFP);                          /* \___ New lines */
      NewLine(wordsFP);                         /* /             */
      
      lineCount++;                              /* read another line */
   }

   fclose(inputFP);                             /* \                 */
   fclose(tagsFP);                              /*  >--- Close files */
   fclose(wordsFP);                              /* /                 */

   printf("Stripped [%s]: %u words, %u lines\n\n",
          inputFile, wordCount, lineCount);
}


tokenList *ReadLineTokens(FILE *ifile, bool *inFile)
{
   char  buffer[MAXSTRING * 4];

   ReadLineText(ifile, buffer, MAXSTRING * 4, inFile);

   return GetTokens(buffer);
}

void ReadLineText(FILE *ifile, char *buffer, unsigned length, bool *inFile)
{
   char        read;                   /* Character read from 'file' */
   unsigned    count;                  /* number of characters in 'buffer' */

   count = 0;

   fread(&read, 1, 1, ifile);

   while (read != EOL && *inFile) {
      if (feof(ifile))
         *inFile = FALSE;

      else {
         *buffer++ = read;
         count++;

         if (count == length - 1)      /* Filled the buffer */
            break;
      }

      fread(&read, 1, 1, ifile);
   }

   *buffer = EOS;
}


/* -- Local functions -- */

/* -- Function NewLine
 * Pre: 'FP' is a file opened for writing
```

```
 * Post: a new line was written to 'FP'
 */
void NewLine(FILE *FP)
{
   fprintf(FP, "\n");
}


/* -- Function MarkEOS
 * Pre: 'FP' is a file opened for writing
 * Post: (options.markEOS == TRUE && EOS marker was written to file) ||
 *       (options.markEOS == FALSE)
 */
void MarkEOS(FILE *FP)
{
   if (options.markEOS)
      fprintf(FP, "%s", options.markEOSTag);
}


/* -- Function MarkExtra
 * Pre: 'FP' is a file opened for writing
 * Post: (options.markExtra == TRUE && Extra tag marker was written to file) ||
 *       (options.markExtra == FALSE)
 */
void MarkExtra(FILE *FP)
{
   if (options.markExtra)
      fprintf(FP, "%s", options.markExtraTag);
}

/* -- Function WriteToken
 * Pre: 'FP' is a file opened for writing
 *      'token' is a valid token
 * Post: 'token' + " " (space) was written to 'FP'
 */
void WriteToken(FILE *FP, tokenList *token)
{
   fprintf(FP, "%s ", token -> token);
}

/* -- Function CheckSequence
 * Pre: 'tagsFP' is a file opened for writing; destination for tags
 *      'wordsFP' is a file opened for writing; destination for words
 *      'tag' is a string to check if it matches a tag
 *      'word' is a string so that if it comes after 'tag' then it must be a word
 *      'index' is an index into a token list
 * Post: If index == 'tag' and index -> NEXT == 'word' then
 *        'tag' --> tags
 *        'word --> words && TRUE is returned
 *       Otherwise 'tag' --> tags && FALSE is returned
 *
 * Note: Due to the stupidity of the WALES corpus designers, many tags were used with the same
 *       names as common words, with no way to identify wether one was a tag or a word.
 */
int CheckSequence(   FILE *tagsFP, FILE *wordsFP,
         string tag, string word,
         tokenList *index)
{
   if ((strcmp(index -> token, tag) == 0) && (strcmp(index -> NEXT -> token, word) == 0)) {
      WriteToken(tagsFP, index);
      WriteToken(tagsFP, index -> NEXT);
      return TRUE;
   } else
      return FALSE;
}


/* --- END of SymStrip.c --- */
```

9.2.5 MakeFSA

Application purpose

MakeFSA
MakeFSA constructs an FSA definition from the probed hidden unit activations from a recurrent Elman network simulated with tlearn. A cluster analysis program is required to extract the locations of the FSA's states within the hidden unit space. The resulting clusters are loaded into MakeFSA. Transition tables are generated for the hidden unit activation data, and used to construct a deterministic FSA.

Usage

```
*** MakeFSA -- write an FSA definition for a tlearn probe run
    Version 0.06 build 11:12:37 Oct  8 1999 [IRIX64 6.5 IP25 mips]
Usage: makefsa {k-means vectors} {pattern file} {probe file}
               {tags file} {FSA definition}
Where: {k-means vectors} -- mean vectors from k-means clustering
       {pattern file}    -- pattern definition of inputs
       {probe file}      -- output from tlearn probe
       {tags file}       -- tags representing input/output lines
                            ['-' for no translation]
       {FSA definition}  -- file to write FSA to ['-' for stdout]
```

Required input files:

- A codebook resulting from either ellipsoidal or spherical cluster analysis
- A pattern file representing the training sequence
- A .probe file generated by a probe run of tlearn
- An optional .reset file used by tlearn during learning

MakeFSA will generate an FSA definition file that can be simulated with Ingo Schellhammer's autosim FSA simulator.

Modes used

StdDefs, 2DArray, TokenLst, TokScan, Vector_Utils, Vector_Read, Gauss, Cluster, CorrMatrix, SmdArray

Source code

Makefile

```
TARGET=makefsa
VERSION=020
CFILES=makefsa.c   vector_utils.c   vector_read.c   tokenlst.c   tokscan.c   smdarray.c   cluster.c
corr_matrix.c gauss2.c 2darray.c
HFILES=stddefs.h   vector_utils.h   vector_read.h   tokenlst.h   tokscan.h   smdarray.h   cluster.h
corr_matrix.h gauss.h 2darray.h
OBJFILES=${TARGET}.o   vector_utils.o   vector_read.o   tokenlst.o   tokscan.o   smdarray.o   cluster.o
corr_matrix.o gauss2.o 2darray.o
DISTFILES=Makefile makefsa.help.make ${CFILES} ${HFILES} ${TARGET}
SYSVER=\"`uname -mprs`\"

CC=cc
CFLAGS=-apo -64 -DPLATFORM="${SYSVER}" -DUNIX
LFLAGS=-lm

${TARGET}: ${OBJFILES}
   ${CC} ${CFLAGS} -o ${TARGET} ${OBJFILES} ${LFLAGS}

help: ${TARGET}
   ${TARGET} --help

clean:
   rm -f *.o ${TARGET}

dist: ${DISTFILES}
```

```
   tar cvf ${TARGET}_v${VERSION}.tar ${DISTFILES}
   gzip -f ${TARGET}_v${VERSION}.tar

.c.o:
   ${CC} ${CFLAGS} -c $*.c
```

MakeFSA.c

```
/* Makefsa -- Generates an FSA from either k-means or ellipsoidal cluster
 *             analysis and tlearn hidden units probe output
 *            Used to make a corresponding FSA for an elman network one step
 *             lookahead task
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 4th October, 1999
 * Modified: 11th September, 2000
 * Version: 0.20
 *
 * Usage: makefsa {cluster type}        -- s (spherical) or e (elliptical)
 *               {cluster means input} -- whitespace-delimited, one vector per line
 *               {pattern file}        -- : (line #) (input) (output)
 *               {probe file}          -- output from tlearn -p run
 *               {tags file}           -- tags representing input / output lines
 *                                        (no translation occurs if '-' is specified)
 *               {resets file}         -- tlearn reset file
 *                                        Will reset FSA to zero state for each reset
 *                                        (no resets are used if '-' is specified)
 *               {FSA definiton file}  -- write FSA definition to this file
 *                                        (writes to stdout if '-' is specified)
 *
 * Modules used: vector_utils, vector_read, cluster, stddefs, smdarray
 *
 * Aknowledgements: Schellhammer, I., Diederich, J., Towsey, M., Brugman, C., "Knowledge Extraction
and Recurrent Neural Networks: An Analysis of an Elman Network trained on a Natural Language
Learning Task"
 */

/* -- Required modules -- */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>
#if defined(UNIX)
   #include <unistd.h>
#endif
#if defined(WIN32)
   #define  strcasecmp  _stricmp
#endif
#include <string.h>
#include "stddefs.h"
#include "vector_utils.h"
#include "vector_read.h"
#include "cluster.h"
#include "2darray.h"
#include "corr_matrix.h"
#include "smdarray.h"
#include "tokenlst.h"
#include "tokscan.h"


/* -- defines -- */

#define  APP_NAME       0
#define  HELP           1
#define  CLUSTER_TYPE   1
#define  CODEBOOK_FILE  2
#define  PATTERN_FILE   3
#define  PROBE_FILE     4
#define  TAGS_FILE      5
#define  RESET_FILE     6
#define  FSA_FILE       7
#define  NUMARGS        8

#define  VERSION        "0.20"
#define  HELP_FILE      "makefsa.help"
#define  MAXTAGLENGTH   20
#define  MAXCLASSES     100         /* Used for unigram array if */
                                    /* no tags file was supplied */
#define  MAX_CLUSTERS   200
```

```
#define  CODEBOOK_KMEANS      vector      **
#define  CODEBOOK_O2CLUST     corrMatrix  *

#define  Abort(x) AbortPrint((x), __LINE__, __FILE__);
#define  MAX(x, y)   ((x) > (y) ? (x) : (y))

/* -- kmeans typedefs -- */

typedef void  *codebook;             /* could be kmeans or o2clust */


/* -- kmeans helper function prototypes -- */

void Usage(FILE *output, char *appName);
void Help(FILE *output, char *appName);
void AbortPrint(int code, int line, char *file);
void ProcessCommandLine(int argc, char *argv[],
         FILE **meansFile, FILE **patternFile, FILE **probeFile,
         FILE **tagsFile, FILE **resetFile, FILE **fsaFile,
         bool *useTags, bool *useResets, bool *useO2);
void CloseFiles(FILE *cbFile, FILE *patternFile, FILE *probeFile, FILE *tagsFile, FILE *resetFile,
FILE *fsaFile);
void PrintBanner(char *argv[], FILE *fsaFile, bool useTags, bool useResets, bool useO2);
void ReadPattern(FILE *patternFile, int *predecessor, int *successor, bool *inFile);
int FindNearestNeighbour(vector *vect, codebook currentBook, bool useO2, int numSeeds);
bool CheckClusterDimensions(cluster *clust);
codebook ReadMeans(FILE *cbFile, bool useO2, int *numClusters);
void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs);
tokenList *ReadWholeFile(FILE *input);
void MakeFSADefHeader(FILE *fsaFile, int rescueState, int rescueOutput, int startState);
int FindStartState(codebook currentBook, bool useO2, int numClusters);
void ZeroUnigramArray(int array[], int size);
int FindMaxSlot(int array[], int size);
int InitResetFile(FILE *resetFile);
int ReadReset(FILE *resetFile);

/* -- Codebook functions -- */
/*codebook CodebookNew(int slots, bool useO2);*/
codebook CodebookDestroy(codebook book, int slots, bool useO2);
codebook CodebookNewFromCluster(cluster *clust, bool useO2);
codebook CodebookMakeCopy(codebook book, int slots, bool useO2);
/*void WriteCodebook(FILE *output, codebook book, int slots, bool useO2);*/


/* -- main -- */

int main(int argc, char *argv[])
{
   FILE     *cbFile, *patternFile, *probeFile, *tagsFile, *resetFile, *fsaFile;
   codebook book;
   int      numClusters, numInputLines, numTeachExamples, numTransitions;
   bool     terminate, useO2;
   vector      *vect;
   int      predecessor, successor, thisState, lastState,
            startState, rescueOutput;
   bool     inPatFile, inProbeFile, useTags, useResets;
   smdarray fsaTransitions;
   char     **tagsArray;
   int      *unigramStats;


/* srand48((double) time(0) + (double) getpid()); /* Initiallise rand48 generator */

   ProcessCommandLine(  argc, argv,
            &cbFile, &patternFile, &probeFile,
            &tagsFile, &resetFile, &fsaFile,
            &useTags, &useResets, &useO2);

   PrintBanner(argv, fsaFile, useTags, useResets, useO2);

   if (useTags) {
      ReadTags(tagsFile, &tagsArray, &numInputLines);
      fprintf(stderr, "Read %d tags (%d input lines).\n", numInputLines, numInputLines);
      if (!(unigramStats = (int *) malloc(numInputLines * sizeof(int)))) {
         fprintf(stderr, "*** Could not allocate inputs unigram array.\n");
         fprintf(stderr, "    ([%d] classes)\n", numInputLines);
         Abort(8);
      }
      ZeroUnigramArray(unigramStats, numInputLines);
   } else {
      if (!(unigramStats = (int *) malloc(MAXCLASSES * sizeof(int)))) {
```

```
         fprintf(stderr, "*** Could not allocate inputs unigram array.\n");
         fprintf(stderr, "    ([%d] classes -- MAXCLASSES)\n", MAXCLASSES);
         Abort(9);
      }
      ZeroUnigramArray(unigramStats, MAXCLASSES);
   }

   if (useO2) {
      book = (CODEBOOK_O2CLUST) ReadMeans(cbFile, useO2, &numClusters);
   } else {
      book = (CODEBOOK_KMEANS) ReadMeans(cbFile, useO2, &numClusters);
   }
   if (numClusters == 0) {
      fprintf(stderr, "*** No clusters to convert!\n");
      Abort(5);
   }
   fprintf(stderr, "Read %d clusters.\n", numClusters);

   fsaTransitions = sArrayCreate();
   InitResetFile(resetFile);

   startState = FindStartState(book, useO2, numClusters);
   lastState = startState;       /* Begin at zero state */
   if (!useTags) numInputLines = 0; /* Count input lines if we weren't given it */
   numTeachExamples = 0;         /* Current training pattern number */
   inPatFile = inProbeFile = TRUE;
   terminate = FALSE;
   while (!terminate) {
      ReadPattern(patternFile, &predecessor, &successor, &inPatFile);
      unigramStats[predecessor]++;  /* Count input */
      if (useResets && ReadReset(resetFile) == numTeachExamples)
         lastState = startState;    /* Reset the FSA */
      if (!useTags) numInputLines = MAX(numInputLines, predecessor);
      if (!(vect = VectorReadLine(probeFile, &inProbeFile)))
         break;

      thisState = FindNearestNeighbour(vect, book, useO2, numClusters) + 1;
      if (!sArrayIncrement(fsaTransitions, 4, 1, lastState, predecessor, thisState, successor)) {
         fprintf(stderr, "*** Error manipulating smdArray.\n");
         fprintf(stderr,  "        Cell:  [%d][%d][%d][%d]\n",  lastState,  predecessor,  thisState,
successor);
         Abort(6);
      }
      if (lastState == 0) startState = thisState;  /* get start state */
      lastState = thisState;
      terminate = !(inPatFile && inProbeFile);
      numTeachExamples++;
      if (((numTeachExamples % 200) == 0) && (numTeachExamples > 1))
         fprintf(stderr, "Read %d inputs...\n", numTeachExamples);
   }

   fprintf(stderr, "Read %d inputs total.\n", numTeachExamples - 1);
   fprintf(stderr, "Read %d transitions total.\n", sArrayCountLeaves(fsaTransitions));

   rescueOutput = FindMaxSlot(unigramStats, numInputLines);
   MakeFSADefHeader(fsaFile, startState, 10, startState);

   numTransitions = 0;

   for (lastState = 1; lastState <= numClusters; lastState++) {
      for (predecessor = 0; predecessor < numInputLines; predecessor++) {
         int   mpState = 0,      /* most probable state */
               mpOutput = 0,
               maxCount = 0,
               count;

         if (!sArrayDoesCellExist(fsaTransitions, 2, lastState, predecessor))
            continue;      /* this transition desn't exist */

         for (thisState = 0; thisState <= numClusters; thisState++) {
            if (sArrayDoesCellExist(fsaTransitions, 3, lastState, predecessor, thisState)) {
               if   ((count   =   sArrayBranchWeight(sArrayGetCell(fsaTransitions,   3,   lastState,
predecessor, thisState))) > maxCount) {
                  maxCount = count;
                  mpState = thisState;
               }
            }
         }

         maxCount = 0;

         for (successor = 0; successor < numInputLines; successor++) {
```

```
            if   ((count   =   sArrayAccess(fsaTransitions,   4,   lastState,   predecessor,   mpState,
successor)) > maxCount) {
               maxCount = count;
               mpOutput = successor;
            }
         }

         if (useTags)
            fprintf(fsaFile,    "%d,%d,%s,%s\n",    lastState,    mpState,    tagsArray[predecessor],
tagsArray[mpOutput]);
         else
            fprintf(fsaFile, "%d,%d,%d,%d\n", lastState, mpState, predecessor, mpOutput);

         numTransitions++;
      }
   }

   fprintf(stderr, "Pruned tree: %d transitions left.\n", numTransitions);

   CloseFiles(cbFile, patternFile, probeFile, tagsFile, resetFile, fsaFile);

   return 0;
}


/* -- helper functions -- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** MakeFSA -- write an FSA definition for a tlearn probe run\n");
   fprintf(output, "    Version %s build %s %s [%s]\n", VERSION, __TIME__, __DATE__, PLATFORM);
   fprintf(output,  "Usage:  %s  {cluster  type}  {codebook  file}  {pattern  file}  {probe  file}\n",
appName);
   fprintf(output, "               {tags file} {reset file} {FSA definition}\n");
   fprintf(output, "Where: {cluster type}    -- either (s)pherical or (e)lliptical\n");
   fprintf(output, "       {codebook file}   -- codebook from a clustering run\n");
   fprintf(output, "       {pattern file}    -- pattern definition of inputs\n");
   fprintf(output, "       {probe file}      -- output from tlearn probe\n");
   fprintf(output, "       {tags file}       -- tags representing input/output lines\n");
   fprintf(output, "                            ['-' for no translation]\n");
   fprintf(output, "       {reset file}      -- tlearn reset file.  Each reset will set the\n");
   fprintf(output, "                            FSA to the 'zero' state\n");
   fprintf(output, "                            ['-' for no resets]\n");
   fprintf(output, "       {FSA definition}  -- file to write FSA to ['-' for stdout]\n");
   fprintf(output, "For detailed info, type %s --help\n", appName);
   fprintf(output, "\n");
}

void AbortPrint(int code, int line, char *file)
{
   fprintf(stderr, "Aborting [%s | %d]...\n", file, line);
   exit(code);
}

void ProcessCommandLine(int argc, char *argv[], FILE **cbFile, FILE **patternFile, FILE **probeFile,
FILE **tagsFile, FILE **resetFile, FILE **fsaFile, bool *useTags, bool *useResets, bool *useO2)
{
   if ((argc > 1) && (strcasecmp(argv[HELP], "--help") == 0)) {
      FILE  *helpFile;

      if (!(helpFile = fopen(HELP_FILE, "wt"))) {
         fprintf(stderr, "*** Could not create help file [%s].\n", HELP_FILE);
         Abort(1);
      }

      Help(helpFile, argv[APP_NAME]);
      printf("Created %s\n", HELP_FILE);
      exit(0);
   }

   if (argc < NUMARGS) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      Abort(2);
   }

   if (argc > NUMARGS)
      fprintf(stderr, "--- Warning: extra arguments ignored.\n");

   if (strcasecmp(argv[CLUSTER_TYPE], "s") == 0) {
      *useO2 = FALSE;
   } else if (strcasecmp(argv[CLUSTER_TYPE], "e") == 0) {
```

```
      *useO2 = TRUE;
   } else {
      fprintf(stderr, "*** Cluster type must be either (s)pherical or (e)lliptical.\n");
      Abort(20);
   }

   if (!(*cbFile = fopen(argv[CODEBOOK_FILE], "rt"))) {
      fprintf(stderr, "*** Could not open cluster means file [%s] for input.\n",
argv[CODEBOOK_FILE]);
      Abort(2);
   }

   if (!(*patternFile = fopen(argv[PATTERN_FILE], "rt"))) {
      fprintf(stderr, "*** Could not open pattern file [%s] for input.\n", argv[PATTERN_FILE]);
      Abort(3);
   }

   if (!(*probeFile = fopen(argv[PROBE_FILE], "rt"))) {
      fprintf(stderr, "*** Could not open probe file [%s] for input.\n", argv[PROBE_FILE]);
      Abort(4);
   }

   if (strcmp(argv[TAGS_FILE], "-") == 0) {
      *useTags = FALSE;
      *tagsFile = NULL;

   } else {
      *useTags = TRUE;
      if (!(*tagsFile = fopen(argv[TAGS_FILE], "rt"))) {
         fprintf(stderr, "*** Could not open tags file [%s] for input.\n", argv[TAGS_FILE]);
         Abort(5);
      }
   }

   if (strcmp(argv[RESET_FILE], "-") == 0) {
      *useResets = FALSE;
      *resetFile = NULL;

   } else {
      *useResets = TRUE;
      if (!(*resetFile = fopen(argv[RESET_FILE], "rt"))) {
         fprintf(stderr, "*** Could not open reset file [%s] for input.\n", argv[TAGS_FILE]);
         Abort(6);
      }
   }

   if (strcmp(argv[FSA_FILE], "-") == 0)
      *fsaFile = stdout;

   else {
      if (!(*fsaFile = fopen(argv[FSA_FILE], "wt"))) {
         fprintf(stderr, "*** Could not open FSA definition file [%s] for output.\n",
argv[FSA_FILE]);
         Abort(7);
      }
   }
}

void CloseFiles(FILE *cbFile, FILE *patternFile, FILE *probeFile, FILE *tagsFile, FILE *resetFile,
FILE *fsaFile)
{
   if (cbFile != NULL) fclose(cbFile);
   if (patternFile != NULL) fclose(patternFile);
   if (probeFile != NULL) fclose(probeFile);
   if (tagsFile != NULL) fclose(tagsFile);
   if (resetFile != NULL) fclose(resetFile);
   if (fsaFile != stdout) fclose(fsaFile);
}


void PrintBanner(char *argv[], FILE *fsaFile, bool useTags, bool useResets, bool useO2)
{
   fprintf(stderr, "- MakeFSA ver %s\n", VERSION);
   fprintf(stderr, " o Clustering type: ");
   useO2 ?
      fprintf(stderr, "ellipsoidal\n") :
      fprintf(stderr, "spherical\n");
   fprintf(stderr, " o Clusters: << [%s]\n", argv[CODEBOOK_FILE]);
   fprintf(stderr, " o Input patterns: << [%s]\n", argv[PATTERN_FILE]);
   fprintf(stderr, " o Hidden units probe: << [%s]\n", argv[PROBE_FILE]);
   fprintf(stderr, " o I/O line tags definitions: << [");
   useTags ?
```

```
      fprintf(stderr, "%s]\n", argv[TAGS_FILE]) :
      fprintf(stderr, "not translated]\n");
   fprintf(stderr, " o Resets: << [");
   useResets ?
      fprintf(stderr, "taken from %s]\n", argv[RESET_FILE]) :
      fprintf(stderr, "not used]\n");
   fprintf(stderr, " o FSA definition: >> [");
   fsaFile == stdout ?
      fprintf(stderr, "- stdout]\n") :
      fprintf(stderr, "%s]\n", argv[FSA_FILE]);
}

void Help(FILE *helpFile, char *appName)
{
   fprintf(helpFile, "-----------------------------------\n");
   fprintf(helpFile, "   Help file for MakeFSA ver %s\n", VERSION);
   fprintf(helpFile, "-----------------------------------\n\n");

   Usage(helpFile, appName);

#include "makefsa.help.make"
}


void ReadPattern(FILE *patternFile, int *predecessor, int *successor, bool *inFile)
{
   int     patternNum;

   fscanf(patternFile, "%d %d %d", &patternNum, predecessor, successor);

   if (feof(patternFile))
      *inFile = FALSE;
}

codebook CodebookNew(int slots, bool useO2)
{
   codebook newBook;

   if (useO2) {
      if (!(newBook = (CODEBOOK_O2CLUST) malloc(slots * sizeof(corrMatrix))))
         return NULL;
   } else {
      if (!(newBook = (vector **) malloc(slots * sizeof(vector *))))
         return NULL;
   }

   return newBook;
}

bool CheckClusterDimensions(cluster *clust)
{
   int     dimensions;             /* dimension to check against */
   cluster *cIndex;                /* index into cluster */

   if (clust == NULL) return TRUE; /* empty cluster, so we pass */

   dimensions = clust -> vect -> dimensions;
   cIndex = clust -> next;         /* the first one passes by definition */

   while (cIndex != NULL) {
      if (cIndex -> vect -> dimensions != dimensions)
         return FALSE;

         cIndex = cIndex -> next;
   }

   return TRUE;
}

codebook CodebookNewFromCluster(cluster *clust, bool useO2)
{
   codebook        newBook;                /* new codebook */
   cluster         *cIndex;                /* index into cluster */
   int             slot;                   /* slot index into codebook */

   if (!(newBook = CodebookNew(ClusterSize(clust), useO2)))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      ((CODEBOOK_KMEANS) newBook)[slot] = VectorMakeCopy(cIndex -> vect);
```

```
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

codebook CodebookDestroy(codebook book, bool useO2, int slots)
{
   int     slot;

   slot = 0;
   while (slot < slots) {
      if(useO2) {
         CMatrixDestroy(((CODEBOOK_O2CLUST) book)[slot]);
      } else {
         VectorDeallocate(((CODEBOOK_KMEANS) book)[slot]);
      }

      slot++;
   }

   free(book);
   return NULL;
}

int FindNearestNeighbour(vector *vect, codebook currentBook, bool useO2, int numSeeds)
{
   int      slot;           /* index into codebook */
   double   minDist,        /* current minimum distance */
            dist;           /* current distance */
   double   minSlot;        /* current slot of minimum distance*/

   minSlot = 0;
   if (useO2) {
      minDist = CMatrixCorrelationWithVect(((CODEBOOK_O2CLUST) currentBook)[0], vect);
   } else {
      minDist = VectorEuclideanDist(vect, ((CODEBOOK_KMEANS) currentBook)[0]);
   }

   slot = 1;
   while (slot < numSeeds) {
      if (useO2) {
         dist = CMatrixCorrelationWithVect(((CODEBOOK_O2CLUST) currentBook)[slot], vect);
      } else {
         dist = VectorEuclideanDist(vect, ((CODEBOOK_KMEANS) currentBook)[slot]);
      }

      if (dist < minDist) {
         minDist = dist;
         minSlot = slot;
      }
      slot++;
   }

   return (int) minSlot;
}


codebook ReadMeans(FILE *cbFile, bool useO2, int *numClusters)
{
   corrMatrix  *book;
   cluster  *clust;

   if (useO2) {
      int   slot;

      if (!(book = (CODEBOOK_O2CLUST) malloc(MAX_CLUSTERS * sizeof(corrMatrix))))
         return NULL;

      for (slot = 0; slot < MAX_CLUSTERS; slot++)
         book[slot] = NULL;

      slot = 0;
      while (!feof(cbFile)) {
         book[slot] = CMatrixRead(cbFile);
         if (book[slot] != NULL)
            slot++;
      }
      *numClusters = slot;
      return book;
```

```
   } else {
      clust = ClusterNew();
      clust = ClusterReadFromFile(cbFile);

      *numClusters = ClusterSize(clust);
      return CodebookNewFromCluster(clust, useO2);
   }
}

tokenList *ReadWholeFile(FILE *input)
{
   tokenList   *list, *temp;
   bool        inFile;

   inFile = TRUE;
   list = ReadLineTokens(input, &inFile);          /* Get the first line */

   if (list == NULL)
      return list;

   while (inFile) {
      temp = ReadLineTokens(input, &inFile);  /* Get the next line */
      list = ConcatenateTokenList(list, temp);
   }

   return list;
}

void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs)
{
   tokenList   *allTags, *index;
   int         numTags;

   allTags = ReadWholeFile(tagsFile);

   if (allTags == NULL) {
      fprintf(stderr, "*** Error reading tags file.\n");
      Abort(7);
   }

   numTags = 0;
   index = allTags;
   while (index != NULL) {
      index = index -> NEXT;
      numTags++;
   }

   *numOutputs = numTags;

   if (!Allocate2DArray(tagsArray, 1, numTags, MAXTAGLENGTH)) {
      fprintf(stderr, "*** Could not allocate tag array.\n");
      Abort(8);
   }

   index = allTags;
   numTags = 0;
   while (index != NULL) {
      strcpy((*tagsArray)[numTags], index -> token);
      index = index -> NEXT;
      numTags++;
   }
   DestroyTokenList(allTags);
}

void MakeFSADefHeader(FILE *fsaFile, int rescueState, int rescueOutput, int startState)
{
   fprintf(fsaFile, "Finite State Automaton Simulator\n");
   fprintf(fsaFile, "Rescue State = %d\n", rescueState);
   fprintf(fsaFile, "Rescue Output = %d\n", rescueOutput);
   fprintf(fsaFile, "Start State = %d\n", startState);
   fprintf(fsaFile, "Transitions:\n");
   fprintf(fsaFile, "(FromState,ToState,Input,Output)\n");
}

int FindStartState(codebook currentBook, bool useO2, int numClusters)
{
   vector   *zero;
   int   startState;

   if (useO2) {
      zero = VectorZero(((CODEBOOK_O2CLUST) currentBook)[0] -> dimensions);
   } else {
```

```
      zoom = VectorZero(((CODEBOOK_KMEANS) currentBook)[0] -> dimensions);
   }

   startState  = FindNearestNeighbour(zero, currentBook, useO2, numClusters);

   zero = VectorDeallocate(zero);
   return startState;
}

void ZeroUnigramArray(int array[], int size)
{
   int   count;

   for (count = 0; count < size; count++)
      array[count] = 0;
}

int FindMaxSlot(int array[], int size)
{
   int   maxSlot = 0,
         maxValue = 0,
         count;

   for (count = 0; count < size; count++) {
      if (array[count] > maxValue) {
         maxValue = array[count];
         maxSlot = count;
      }
   }

   return maxSlot;
}

int InitResetFile(FILE *resetFile)
{
   int   numResets;

   fscanf(resetFile, "%d\n", &numResets);
   return numResets;
}


int ReadReset(FILE *resetFile)
{
   int   reset;

   fscanf(resetFile, "%d\n", &reset);
   return reset;
}


/* --- END of makefsa.c --- */
```

makefsa.help.make

```
/* Help make file for MakeFSA
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au
 * Date: 4th October, 1999
 * Modified: 13th September, 2000
 * Version: 0.02
 */

#define p(x)   fprintf(helpFile, x);

p("\n");
p("This program creates a deterministic FSA from a set of state definitions\n");
p("(clustering codebook) and a set of training vectors.  This is used to obtain an\n");
p("FSA from a tlearn hidden units probe run and an AVQ (Euclidean distance or correlation\n");
p("distance) analysis of the hidden unit states.\n");
p("\n");
p("MakeFSA will generate a state for each cluster in the AVQ analysis.  The\n");
p("training vectors will be read and a transition table generated from state to\n");
p("state given each input.  Non-deterministic transitions are culled by taking\n");
p("the most probable state moved to for each input, as well as the most probable\n");
p("output for each transition.\n");
p("\n");
p("The FSA definition is augmented by a start state and a rescue state, as well as\n");
p("a rescue output.  These are defined as:\n");
p(" o Start state: the state corresponding to the cluster containing the zero\n");
p("   vector.\n");
```

```
p(" o Rescue state: at this stage, this is the same as the start state.\n");
p(" o Rescue output: the most common output for all inputs (i.e. a noun in\n");
p("   language data).\n");
p("\n");
p("Inputs are:\n");
p(" o The type of clustering that was performed (spherical or elliptical).\n");
p(" o A file corresponding to the clusters identified by the AVQ analysis.\n");
p(" o A pattern file with a list of input/output pairs for the training data.\n");
p(" o A file of vectors containing the training data (hidden unit activations)\n");
p(" o An optional file specifing names ('tags') to give to each of the\n");
p("   intput / output lines (tags file).\n");
p("\n");
p("\n");
p("Input formats:\n");
p("Vector files(k-means clusters, training set):\n");
p(" o Vectors are input one per line.\n");
p(" o Vector components are separated by whitespace (space or tab).\n");
p(" o All vectors in the file should be of the same dimensionality.\n");
p("\n");
p("Codebook file:\n");
p(" o For k-means clusters, see above.\n");
p(" o For o2clust clusters, the file format is determined by CMatrixWrite in the");
p("   corr_matrix.c source file.\n");
p("\n");
p("Pattern file:\n");
p(" o each line follows the same pattern: input# intput output.\n");
p(" o input# starts from zero and counts up to the number of inputs.\n");
p(" o input and output represent the input and output lines that should be\n");
p("   activated.\n");
p("\n");
p("Tags file:\n");
p(" o Tags are separated by whitespace (space, tab or <cr>).\n");
p(" o If this file is given, there MUST be at least as many tags as input / output\n");
p("   lines.  The program's behavious is undefined otherwise.\n");
p("\n");
p("\n");
p("Output formats:\n");
p("FSA Definition:\n");
p(" o The output format is designed to be used in Ingo Schellhammer's FSA\n");
p("   simulator 'autosim'.  It consists of an explicit header which must be\n");
p("   character perfect, containing the start state, rescue state and rescue\n");
p("   output.\n");
p(" o Deterministic FSA transitions are given one per line, separated by commas.\n");
p(" o If a tags file was specifed, inputs and outputs are translated into tags.\n");
p("   (Note that this format is not compatible with 'autosim').\n");
p("\n");
p("\n");
p("Note: MakeFSA was designed for use on an elman-type recurrent neural network\n");
p("      trained on language data to perform a one-step-lookahead task.  Therefore\n");
p("      the input and output lines correspond one-to-one, and the tags file can\n");
p("      be used to describe both.  This should hold true for most FSAs. The\n");
p("      pattern files used were generated by concatenating lines in tlearn\n");
p("      ___.teach and ___.data files.  This will only work if the tlearn files\n");
p("      are in locallist format.\n");
p("\n");
p("\n");
p("\n");
p("\n");

#undef p

/* --- END of makefsa.help.make --- */
```

9.2.6 pattern

Application purpose

pattern generates .pattern files from tlearn .teach and .data files. .pattern files are used in several other applications, such as dstat and MakeFSA.

Usage

```
***pattern ver 0.01 -- creates pattern file from tlearn data
Usage: pattern {baseFileName}
       pattern {dataFile} {teachFile} {patternFile}
```

pattern can take as arguments either a base filename to both read the tlearn data from and to write the .pattern file to, or three separate filenames.

Modules used

StdDefs, TokenLst, TokScan,

Source code

Makefile

```
DISTFILES=pattern.c tokscan.c tokscan.h tokenlst.c tokenlst.h pattern
CC=gcc -g

pattern: pattern.o tokscan.o tokenlst.o
   ${CC} -o pattern pattern.o tokscan.o tokenlst.o
clean:
   rm -f *.o pattern
dist:
   tar cvf pattern_v001.tar $(DISTFILES)
   gzip pattern_v001.tar

.c.o:
   ${CC} -c $*.c
```

pattern.c

```
/* -- Pattern.c -- make a .pattern file from tlearn data files
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG August 1999
 * Date: 9th August, 1999
 * Modified:
 * Version: 0.01
 */

/* --- Defines --- */

#define  VERSION        "0.01"
#define  APPNAME        0
#define  BASEFILENAME   1
#define  DATAFILE       1
#define  TEACHFILE      2
#define  PATTERNFILE    3
#define  NUMARGS        4
#define  MAX_STRING     512

#define  MIN(x, y)   ((x) > (y) ? (y) : (x))

/* --- Included headers --- */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "tokenlst.h"
#include "tokscan.h"
#include "stddefs.h"


/* --- Enums --- */

typedef enum fileTypeEnum {distributed, localist, undefined} fileType;
```

```
/* --- Helper functions --- */

void InterpretCommandLine(int argc, char *argv[]);
void Usage(FILE *output, char *argv[]);
void OpenFiles(int argc, char *argv[], FILE **dataFile, FILE **teachFile, FILE **patternFile);
void Abort(int errcode);
void CloseFiles(FILE *dataFile, FILE *teachFile, FILE *patternFile);
void ProcessFiles(FILE *dataFile, FILE *teachFile, FILE *patternFile);
fileType DetermineType(char *text);
void ReadFileInfo(FILE *file, fileType *fileStruct, int *numPatterns);
char *FileTypeString(fileType file);
void EatLine(FILE *file);


/* -- Main -- */

void main(int argc, char *argv[])
{
   FILE  *dataFile,
      *teachFile,
      *patternFile;

   InterpretCommandLine(argc, argv);
   OpenFiles(argc, argv, &dataFile, &teachFile, &patternFile);

   ProcessFiles(dataFile, teachFile, patternFile);

   CloseFiles(dataFile, teachFile, patternFile);
}


/* --- Helper functions --- */

void InterpretCommandLine(int argc, char *argv[])
{
   switch (argc) {
      case 1:
      case 3:
         Usage(stderr, argv);
         fprintf(stderr, "*** Incorrect number of arguments.\n");
         Abort(1);
         break;

      default:
         break;
   }

   if (argc > NUMARGS)
      fprintf(stderr, "--- WARNING: extra arguments ignored.\n");
}

void Usage(FILE *output, char *argv[])
{
   fprintf(output, "\n***pattern ver %s -- creates pattern file from tlearn data\n", VERSION);
   fprintf(output, "Usage: %s {baseFileName}\n", argv[APPNAME]);
   fprintf(output, "       %s {dataFile} {teachFile} {patternFile}\n", argv[APPNAME]);
}

void OpenFiles(int argc, char *argv[], FILE **dataFile, FILE **teachFile, FILE **patternFile)
{
   char  buffer[MAX_STRING],
      dataFName[MAX_STRING],
      teachFName[MAX_STRING],
      patternFName[MAX_STRING];

   switch(argc) {
      case 2:     /* Base name specified only */
         strcpy(dataFName, argv[BASEFILENAME]);
         strcpy(teachFName, argv[BASEFILENAME]);
         strcpy(patternFName, argv[BASEFILENAME]);

         strcat(dataFName, ".data");
         strcat(teachFName, ".teach");
         strcat(patternFName, ".pattern");

         break;

      case 4:
         strcpy(dataFName, argv[DATAFILE]);
         strcpy(teachFName, argv[TEACHFILE]);
```

```
         strcpy(patternFName, argv[PATTERNFILE]);

         break;

      default:
         fprintf(stderr, "*** Undefined error condition.\n");
         Abort(2);
   }

   printf(" - Pattern ver %s\n", VERSION);
   printf(" | Data file: [%s]\n", dataFName);
   printf(" | Teach file: [%s]\n", teachFName);
   printf(" +-> Pattern file: [%s]\n\n", patternFName);


   if ((*dataFile = fopen(dataFName, "rt")) == NULL) {
      fprintf(stderr, "*** Could not open [%s] for input.\n", dataFName);
      Abort(3);
   }

   if ((*teachFile = fopen(teachFName, "rt")) == NULL) {
      fprintf(stderr, "*** Could not open [%s] for input.\n", teachFName);
      fclose(*dataFile);
      Abort(3);
   }

   if ((*patternFile = fopen(patternFName, "wt")) == NULL) {
      fprintf(stderr, "*** Could not open [%s] for output.\n", patternFName);
      fclose(*dataFile);
      fclose(*teachFile);
      Abort(3);
   }
}

void Abort(int errcode)
{
   fprintf(stderr, "Aborting...\n");
   exit(errcode);
}

void CloseFiles(FILE *dataFile, FILE *teachFile, FILE *patternFile)
{
   if (dataFile != NULL)
      fclose(dataFile);

   if (teachFile != NULL)
      fclose(teachFile);

   if (patternFile != NULL)
      fclose(patternFile);
}


void ProcessFiles(FILE *dataFile, FILE *teachFile, FILE *patternFile)
{
   char     buffer[MAX_STRING];
   tokenList   *tokens;
   bool     inFile = TRUE;
   fileType dFileStruct, tFileStruct;
   int      dNumPatterns, tNumPatterns,
         patternIndex;

   ReadFileInfo(dataFile, &dFileStruct, &dNumPatterns);
   ReadFileInfo(teachFile, &tFileStruct, &tNumPatterns);

   printf(  "Data file: type[%s] patterns[%d]\n",
      FileTypeString(dFileStruct), dNumPatterns);
   printf(  "Teach file: type[%s] patterns[%d]\n",
      FileTypeString(tFileStruct), tNumPatterns);

   if (!(dFileStruct == localist && tFileStruct == localist)) {
      fprintf(stderr, "*** Error: file creation supported for localist files only.\n");
      Abort(5);
   }

   if (dNumPatterns > tNumPatterns)
      fprintf(stderr, "--- WARNING: extra patterns in .data will be ignored.\n");
   else if (dNumPatterns < tNumPatterns)
      fprintf(stderr, "--- WARNING: extra patterns in .teach will be ignored.\n");

   patternIndex =  0;
```

```
   while (  patternIndex < MIN(dNumPatterns, tNumPatterns) &&
      !feof(dataFile) && !feof(teachFile)) {

      fprintf(patternFile, "%d ", patternIndex);

      ReadLineText(dataFile, buffer, MAX_STRING, &inFile);
      fprintf(patternFile, "%s ", buffer);

      ReadLineText(teachFile, buffer, MAX_STRING, &inFile);
      fprintf(patternFile, "%s\n", buffer);

      patternIndex++;

      if ((patternIndex % 1000 == 0) && patternIndex > 10)
         printf("Processed %d patterns...\n", patternIndex);
   }

   if (patternIndex > 1000)
      printf("Processed %d patterns total.\n", patternIndex);
}

void ReadFileInfo(FILE *file, fileType *fileStruct, int *numPatterns)
{
   char  buffer[MAX_STRING];

   fscanf(file, "%s", buffer);
   EatLine(file);
   *fileStruct = DetermineType(buffer);

   fscanf(file, "%d", (int *) buffer);
   EatLine(file);
   *numPatterns = *(int *) buffer;
}


fileType DetermineType(char *text)
{
   if (strcmp(text, "distributed") == 0)
      return distributed;

   else if (strcmp(text, "localist") == 0)
      return localist;

   else
      return undefined;
}

char *FileTypeString(fileType file)
{
   switch(file) {
      case distributed:
         return "distributed";

      case localist:
         return "localist";

      case undefined:
         return "undefined";

      default:
         fprintf(stderr, "*** Uncategoriesed enum error.\n");
         Abort(4);
   }
}

void EatLine(FILE *file)
{
   char  read;

   read = 0x0;
   while (read != EOL && !feof(file))
      fread(&read, 1, 1, file);
}

/* --- END of pattern.c --- */
```

9.2.7 vector

Application purpose

vector takes a data set of known tags and sentence boundaries, and writes the corresponding tlearn .teach and .data files. The output can be written in both localist and distributed representations, and the input and output lines can be either binary or normalised together.

Usage

```
*** vector ver 0.21 -- creates tlearn vector files from tag input
Usage: vector {tagfile} {inputfile} {inputType} {outputType} {fileStructure}
{basefilename}
Where: {i/oType} == [a]verage or [b]inary
       {fileStructure} == [l]ocalist or [d]istributed
       {outputfiles} -> basefilename.cf
                        basefilename.data
                        basefilename.teach
                        basefilename.reset
```

The data set consists of a sequence of whitespace-delimited tags, with optional resets denoted by '/C' within the data set.

example.txt

```
HN M H AJT *ADJ & /C
F HN M HN /S /C
ADJ *H H FI *P H FI *P /C
HN M P DD H /C
HN M P M P /C
F HN OM P DD /C
```

The file to define the tags is in the standard format: tokens separated by whitespace.

wales.claire.tags

```
&
ADJ
AJT
AL
CP
DD
DOR
DQ
F
FI
FR
G
H
HN
IE
INF
M
MOC
MOD
N
O
OM
OX
P
Q
SC
ST
T
VO
WH
/S
```

Modules used

StdDefs, TokenLst, TokScan, HTable,

Source code

Makefile

```
DISTFILES=tokenlst.c   tokenlst.h   vector.c   htable.c   htable.h   stddefs.h   test.tags   test.txt
wales.claire.tags 6yrs.5.tag.mark vector
OBJFILES=tokenlst.o vector.o tokscan.o htable.o
VERSION=021
TARGET=vector
CC=gcc

${TARGET}: ${OBJFILES}
   ${CC} -o ${TARGET} ${OBJFILES}

clean:
   rm -f *.o vector
dist: ${DISTFILES}
   tar cvf ${TARGET}_v${VERSION}.tar $(DISTFILES)
   gzip -f ${TARGET}_v${VERSION}.tar

htable.o: htable.c

tokenlst.o: tokenlst.c

vector.o: vector.c

tokscan.o: tokscan.c

.c.o:
   ${CC} -c -g $*.c
```

vector.c

```
/* vector.c -- creates a set of vectors for tlearn
 *
 * Author: Dylan Muir (dr.muir@qut.edu.au)
 * Created: 17th May, 1999
 * Modified: 8th October, 1999
 * Version: 0.21
 */

#include <stdio.h>
#include <string.h>
#include "stddefs.h"
#include "htable.h"
#include "tokenlst.h"
#include "tokscan.h"


/* --- Vector defines --- */

#define  BINARY   0
#define  AVERAGE  1

#define  TAGFILE     1
#define  INPUT_TYPE  3
#define  OUTPUT_TYPE 4
#define  INPUT_FILE  2
#define  BASEFILE    6
#define  FILE_STRUCT 5

#define  EXTRAmark   '*'
#define  RESET       "/C"

#define  VERSION        "0.21"
#define  DISTFILETYPE   "distributed\n"      /* Type of output tlearn file */
#define  LOCALFILETYPE  "localist   \n"      /* MUST BE SAME LENGTH */


/* --- Vector enums --- */

typedef enum {binary, averaged} vectorType;
typedef enum {localist, distributed} fileType;


/* --- Helper functions prototypes --- */

void        Usage(FILE *output, char *argv[]);
void        Help(char *argv[]);
tokenList   *ReadWholeFile(FILE *input);
```

```
void        PrintTokenList(FILE *output, tokenList *list);
bool        SetupHashtable(tokenList *tags);
int         CountTokens(tokenList *list);
int         DoVectors(FILE *input, int vectorSize, vectorType intype, vectorType outtype, fileType
fileStruct, FILE *dvFile, FILE *tvFile, FILE *rvFile);
bool        IsExtra(tokenList *tag);
bool        IsReset(tokenList *tag);
void        AddToVector(float vector[], int vectorSize, char *tag);
void        OutputVector(FILE  *output,  float  vector[],  int  vectorSize,  vectorType  printType,
fileType fileStruct);
void        CalcVector(float vector[], int vectorSize);
void        ResetVector(float vector[], int vectorSize);
void        OpenFiles(  int argc, char *argv[],
                        FILE **tagFile, FILE **inputFile,
                        FILE **cfFile, FILE **dvFile, FILE **tvFile, FILE **rvFile);
void        InterpretCommandLine(int  cargc,  char  *argv[],  vectorType  *inputType,  vectorType
*outputType, fileType *fileStruct);
void        OutputReset(FILE *resetFile, int vectorNum);
void        CopyVector(float dest[], float source[], int vectorSize);
void        MakeCfFile(FILE *cfFile, int vectorSize);
void        CloseFiles(FILE *inFile, FILE *cfFile, FILE *dvFile, FILE *tvFile, FILE *rvFile);
void        FixDatFiles(FILE *dvFile, FILE *tvFile, FILE *rvFile, int numVectors, int numResets);
tokenList   *ReadWholeVector(FILE *input, tokenList *index, tokenList **line, float vector[], int
vectorSize, bool *inFile);

/* --- Main function --- */

int main(int argc, char *argv[])
{
   FILE  *tagFile,
         *inputFile,
         *cfFile,
         *dvFile,
         *tvFile,
         *rvFile;

   tokenList   *tags;
   int         vectorSize, numVectors;
   vectorType  inputType, outputType;
   fileType    fileStruct;

   InterpretCommandLine(argc, argv, &inputType, &outputType, &fileStruct);

   OpenFiles(argc, argv, &tagFile, &inputFile, &cfFile, &dvFile, &tvFile, &rvFile);

   tags = ReadWholeFile(tagFile);
   if ((vectorSize = CountTokens(tags)) == 0) {
      fprintf(stderr, "*** Read zero tags from tags file [%s].\n", argv[TAGFILE]);
      fprintf(stderr, "    Cannot continue with tlearn training vectors of size zero.\n");
      fprintf(stderr, "Aborting...\n");
      exit(21);
   }

   printf("Read %d tags\n", vectorSize);

   if (!SetupHashTable(tags)) {
      fprintf(stderr, "*** Cannot set up lookup table for tags.\nAborting...\n");
      exit(1);
   }

   numVectors = DoVectors( inputFile, vectorSize,
                           inputType, outputType,
                           fileStruct,
                           dvFile, tvFile, rvFile);
   MakeCfFile(cfFile, vectorSize);

   printf("Wrote %d vectors total.\nFinished!\n", numVectors);
   CloseFiles(inputFile, cfFile, dvFile, tvFile, rvFile);
}


/* --- Helper function bodies --- */

void Usage(FILE *output, char *argv[])
{
   fprintf(output, "\n*** vector ver %s -- creates tlearn vector files from tag input\n", VERSION);
   fprintf(output,  "Usage:  %s  {tagfile}  {inputfile}  {inputType}  {outputType}  {fileStructure}
{basefilename}\n", argv[0]);
   fprintf(output, "Where: {i/oType} == [a]verage or [b]inary\n");
   fprintf(output, "       {fileStructure} == [l]ocalist or [d]istributed\n");
   fprintf(output, "       {outputfiles} -> basefilename.cf\n");
   fprintf(output, "                        basefilename.data\n");
```

```
   fprintf(output, "                        basefilename.teach\n");
   fprintf(output, "                        basefilename.reset\n");
   fprintf(output, "For detailed info, type %s --help\n\n", argv[0]);
}

void Help(char *argv[])
{
   FILE  *helpFile;

   if ((helpFile = fopen("vector.help", "wt")) == NULL) {
      fprintf(stderr, "\n*** Unable to create \"vector.help\"\n");
      helpFile = stdout;
   }

   fprintf(helpFile, "\n--------------------------------------------\n");
   fprintf(helpFile, " Help file for vector ver %s\n", VERSION);
   fprintf(helpFile, "--------------------------------------------\n");

   Usage(helpFile, argv);

   fprintf(helpFile, "Vector will translate a dataset of tags into vectors for\n");
   fprintf(helpFile, "traning and testing with tlearn.  Inputs are:\n");
   fprintf(helpFile, "   * a file with all the tags contained in the dataset,\n");
   fprintf(helpFile, "     separated by whitespace\n");
   fprintf(helpFile, "   * an input file with resets marked with \"/C\"\n");
   fprintf(helpFile, "   * whether the inputs and outputs are binary (0/1)\n");
   fprintf(helpFile, "     or averaged (sum to 1)\n");
   fprintf(helpFile, "   * whether output files are to be localist or distributed\n");
   fprintf(helpFile, "     (see tlearn documentation for details)\n");
   fprintf(helpFile, "   * a base file name to use\n\n");

   fprintf(helpFile, "Outputs are:\n");
   fprintf(helpFile, "   * [basefilename].cf -- contains skeleton config file\n");
   fprintf(helpFile, "   * [basefilename].data -- contains data (input) vectors\n");
   fprintf(helpFile, "   * [basefilename].teach -- contains teaching (output) vectors\n");
   fprintf(helpFile, "   * [basefilename].reset -- contains resets\n\n");

   fprintf(helpFile, "Using the output files:\n");
   fprintf(helpFile, "Outputs are specified either localist or distributed.\n");
   fprintf(helpFile, "Note that localist is only valid if binary outputs are\n");
   fprintf(helpFile, "specified.  The \".cf\" file will have to be filled out for\n");
   fprintf(helpFile, "the specific network to be trained.\n\n");

   fprintf(helpFile, "Interpreting the vectors:\n");
   fprintf(helpFile, "The vectors are created in the same order as the input tags were\n");
   fprintf(helpFile, "presented to the program.\n");

   fprintf(stderr, "Created the file \"vector.help\"\n\n");
}


tokenList *ReadWholeFile(FILE *input)
{
   tokenList   *list, *temp;
   bool        inFile;

   inFile = TRUE;

   list = ReadLineTokens(input, &inFile);    /* Get the first line */

   if (list == NULL)
      return list;

   while (inFile) {
      temp = ReadLineTokens(input, &inFile); /* Get the next line */
      list = ConcatenateTokenList(list, temp);
   }

   return list;
}

void PrintTokenList(FILE *output, tokenList *list)
{
   while (list != NULL) {
      fprintf(output, "%s ", list -> token);

      list = list -> NEXT;
   }
}

bool SetupHashTable(tokenList *tags)
{
```

```
   int   index = 0;

   if (!hashInitTable(CountTokens(tags)) * 2)
      return FALSE;

   while (tags != NULL) {
      if (!hashInsert(tags -> token, index))
         fprintf(stderr, "--- Duplicate tag: [%s]\n", tags -> token);

      index++;
      tags = tags -> NEXT;
   }

   return TRUE;
}

int CountTokens(tokenList *list)
{
   int   count = 0;

   while (list != NULL) {
      list = list -> NEXT;
      count++;
   }
   return count;
}

int DoVectors(FILE *input, int vectorSize, vectorType intype, vectorType outtype, fileType
fileStruct, FILE *dvFile, FILE *tvFile, FILE *rvFile)
{
   float       *dvector, *tvector;
   tokenList   *line,
               *index;
   bool        inFile;
   int         vectorNum,
               resetNum;


   if ((dvector = (float *) malloc(vectorSize * sizeof(float))) == NULL) {
      fprintf(stderr, "*** Cannot allocate data vector.\nAborting...\n");
      exit(1);
   }

   if ((tvector = (float *) malloc(vectorSize * sizeof(float))) == NULL) {
      fprintf(stderr, "*** Cannot allocate teach vector.\nAborting...\n");
      exit(1);
   }

   if (fileStruct == localist) {
      fprintf(dvFile, LOCALFILETYPE);
      fprintf(tvFile, LOCALFILETYPE);
   } else {
      fprintf(dvFile, DISTFILETYPE);
      fprintf(tvFile, DISTFILETYPE);
   }

   fprintf(dvFile, "           \n");   /* \                                                  */
   fprintf(tvFile, "           \n");   /*  >--- Make space for numbers of vectors, resets, etc. */
   fprintf(rvFile, "           \n");   /* /                                                  */

   vectorNum = 0;
   inFile = TRUE;
   ResetVector(dvector, vectorSize);
   OutputReset(rvFile, 0);
   resetNum = 1;

   index = ReadWholeVector(input, NULL, &line, dvector, vectorSize, &inFile);

   do {
      if (IsReset(index)) {
         OutputReset(rvFile, vectorNum);
         ResetVector(dvector, vectorSize);
         index = index -> NEXT;
         index = ReadWholeVector(input, index, &line, dvector, vectorSize, &inFile);
         resetNum++;
      }

      index = ReadWholeVector(input, index, &line, tvector, vectorSize, &inFile);
      if (index == NULL && !inFile)
         continue;

      if (intype == averaged) CalcVector(dvector, vectorSize);
```

```
      if (outtype == averaged) CalcVector(tvector, vectorSize);
      OutputVector(dvFile, dvector, vectorSize, intype, fileStruct);
      OutputVector(tvFile, tvector, vectorSize, outtype, fileStruct);
      CopyVector(dvector, tvector, vectorSize);

      if ((vectorNum % 1000) == 0 && vectorNum > 0)
         printf("Wrote %d vectors...\n", vectorNum);

      vectorNum++;
   } while (inFile);

   FixDatFiles(dvFile, tvFile, rvFile, vectorNum, resetNum);
   return vectorNum;
}


bool IsExtra(tokenList *tag)
{
   return ((tag != NULL) && (*(tag -> token) == EXTRAmark));
}

bool IsReset(tokenList *tag)
{
   return ((tag != NULL) && (strcmp(tag -> token, RESET) == 0));
}

void ResetVector(float vector[], int vectorSize)
{
   while (vectorSize >= 0) {
      vector[vectorSize] = 0.0;
      vectorSize--;
   }
}

void AddToVector(float vector[], int vectorSize, char *tag)
{
   if (!hashIsIn(tag)) {
      fprintf(stderr, "--- Tag [%s] not defined.\n", tag);
      return;
   }

   vector[hashValue(tag)] = 1.0;
}

void CalcVector(float vector[], int vectorSize)
{
   int   entries,
      index;
   float value;

   index = entries = 0;

   while (index < vectorSize) {
      if (vector[index] != 0.0)
         entries++;
      index++;
   }

   value = 1.0 / entries;

   index = 0;
   while (index < vectorSize) {
      if (vector[index] != 0.0)
         vector[index] = value;
      index++;
   }
}


void OutputVector(FILE *output, float vector[], int vectorSize, vectorType printType, fileType
fileStruct)
{
   int   index = 0,
         notFirst = 0;

   while (index < vectorSize) {
      if (printType == averaged)
         fprintf(output, "%4f ", vector[index]);

      else {
         if (fileStruct == localist) {
         if (((int) vector[index] != 0)) {
```

```
            if (notFirst != 0)
               fprintf(output, ",");
            fprintf(output, "%d", index + 1);
            notFirst = 1;
         }} else
            fprintf(output, "%d ", (int) vector[index]);
      }

      index++;
   }
   fprintf(output, "\n");
}


void OpenFiles(int argc, char *argv[],
               FILE **tagFile, FILE **inputFile,
               FILE **cfFile, FILE **dvFile, FILE **tvFile, FILE **rvFile)
{
   char  buffer[MAXSTRING];

   if (strlen(argv[BASEFILE]) > MAXSTRING - 10) {
      fprintf(stderr,  "***  MAXSTRING  too  short  for  base  filename.\nPlease  recompile.
Aborting...\n");
   }

   if ((*tagFile = fopen(argv[TAGFILE], "rt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to read tags.\nAborting...\n", argv[TAGFILE]);
      exit(1);
   }

   if ((*inputFile = fopen(argv[INPUT_FILE], "rt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to read input.\nAborting...\n", argv[INPUT_FILE]);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".cf");
   if ((*cfFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write config file.\nAborting...\n", buffer);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".data");
   if ((*dvFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write data file.\nAborting...\n", buffer);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".teach");
   if ((*tvFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write teach file.\nAborting...\n", buffer);
      exit(1);
   }

   strcpy(buffer, argv[BASEFILE]);
   strcat(buffer, ".reset");
   if ((*rvFile = fopen(buffer, "wt")) == NULL) {
      fprintf(stderr, "*** Cannot open [%s] to write reset file.\nAborting...\n", buffer);
      exit(1);
   }
}


void  InterpretCommandLine(int  argc,  char  *argv[],  vectorType  *inputType,  vectorType  *outputType,
fileType *fileStruct)
{
   if ((argc > 1) && (strcmp(argv[1], "--help") == 0)) {
      Help(argv);
      exit(0);
   }

   if (argc < 7) {
      Usage(stderr, argv);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      exit(1);
   }

   if (argc > 7)
      fprintf(stderr, "*** WARNING: extra arguments ignored.\n");
```

```
   switch (*argv[INPUT_TYPE]) {
      case 'A':
      case 'a':
         *inputType = averaged;
         break;

      case 'B':
      case 'b':
         *inputType = binary;
         break;

      default:
         Usage(stderr, argv);
         fprintf(stderr, "*** Error in 'input type' field\n");
         exit(1);
   }

   switch (*argv[OUTPUT_TYPE]) {
      case 'A':
      case 'a':
         *outputType = averaged;
         break;

      case 'B':
      case 'b':
         *outputType = binary;
         break;

      default:
         Usage(stderr, argv);
         fprintf(stderr, "*** Error in 'output type' field\n");
         exit(1);
   }

   switch (*argv[FILE_STRUCT]) {
      case 'L':
      case 'l':
         if (!((*inputType == binary) && (*outputType == binary))) {
            Usage(stderr, argv);
            fprintf(stderr, "*** Error in 'file type' field\n");
            fprintf(stderr, "    For localist files, input and output must be 'binary'\n");
            exit(1);
         }
         *fileStruct = localist;    /* localist */
         break;

      case 'D':
      case 'd':
         *fileStruct = distributed;    /* distributed */
         break;

      default:
         Usage(stderr, argv);
         fprintf(stderr, "*** Error in 'file structure' field\n");
         exit(1);
   }

   printf("- Vector ver %s\n", VERSION);
   printf(" Tags file: [%s]\n", argv[TAGFILE]);
   printf(" Input file: [%s]\n", argv[INPUT_FILE]);
   printf(" Output files: [%s].cf, .data, .teach, .reset\n", argv[BASEFILE]);

   printf(" I/O type: ");
   *inputType == averaged ? printf("averaged / ") : printf("binary / ");
   *outputType == averaged ? printf("averaged\n") : printf("binary\n");

   printf(" File structure: ");
   *fileStruct == localist ? printf("localist\n") : printf("distributed\n");
}

void OutputReset(FILE *resetFile, int vectorNum)
{
   fprintf(resetFile, "%d\n", vectorNum);
}

void CopyVector(float dest[], float source[], int vectorSize)
{
   int   index = 0;

   while (index < vectorSize) {
      dest[index] = source[index];
      index++;
```

```
   }
}

void MakeCfFile(FILE *cfFile, int vectorSize)
{
   fprintf(cfFile, "NODES:\n");
   fprintf(cfFile, "nodes = ***\n");
   fprintf(cfFile, "inputs = %d\n", vectorSize);
   fprintf(cfFile, "outputs = %d\n", vectorSize);
   fprintf(cfFile, "output nodes are <node-list>\n");
   fprintf(cfFile, "\nCONNECTIONS:\n");
   fprintf(cfFile, "groups = ***\n");
   fprintf(cfFile, "\nSPECIAL:\n");
}

void CloseFiles(FILE *inFile, FILE *cfFile, FILE *dvFile, FILE *tvFile, FILE *rvFile)
{
   fclose(inFile);
   fclose(cfFile);
   fclose(dvFile);
   fclose(tvFile);
   fclose(rvFile);
}

void FixDatFiles(FILE *dvFile, FILE *tvFile, FILE *rvFile, int numVectors, int numResets)
{
   fseek(dvFile, strlen(DISTFILETYPE), SEEK_SET);
   fseek(tvFile, strlen(DISTFILETYPE), SEEK_SET);
   fseek(rvFile, 0, SEEK_SET);

   fprintf(dvFile, "%d", numVectors);
   fprintf(tvFile, "%d", numVectors);
   fprintf(rvFile, "%d", numResets);
}

tokenList *ReadWholeVector(FILE *input, tokenList *index, tokenList **line, float vector[], int
vectorSize, bool *inFile)
{
   while (index == NULL && *inFile)
      index = *line = ReadLineTokens(input, inFile);

   if (index == NULL && !*inFile)
      return NULL;

   ResetVector(vector, vectorSize);

   AddToVector(vector, vectorSize, index -> token);
   index = index -> NEXT;

   do {
      while (index != NULL) {
         if (IsExtra(index))
            AddToVector(vector, vectorSize, index -> token + 1);
         else
            return index;

         index = index -> NEXT;
      }
      *line = DestroyTokenList(*line);
      index = *line = ReadLineTokens(input, inFile);
   } while (*inFile);

   return index;
}


/* --- END of vector.c --- */
```

9.2.8 tlbe

Application purpose

tlbe stands for **tl**earn **b**it **e**rror. It can extract the true error (not the averaged error generated by tlearn) for a distributed output and target. It will give the number of incorrect predictions over a tlearn run for a one-step-lookahead task.

Usage

```
*** TlearnBitError ver 0.24 -- outputs learning error statistics from tlearn
Usage: tlbe {tagsFile} {inputFile} {patternFile} {outputFile}
Where: {tagsFile}    -- tags representing input lines
       {inputFile}   -- output from tlearn verify sweep
                        (specify '-' for stdin)
       {patternFile} -- file to specify output targets
       {outputFile}  -- file to append output to
                        format:  (#errors)  (av.  bit  error)  (av.  correct)  (av.
output sum)
                        (specify '-' for stdout)
```

Modules used

StdDefs, TokenLst, TokScan

Source code

Makefile

```
DISTFILES=tokenlst.c tokenlst.h tokscan.c tokscan.h tlbe.c tlbe Makefile stddefs.h
OBJFILES=tokenlst.o tokscan.o tlbe.o
TARGET=tlbe
VERSION=024

CC=gcc -g
CFLAGS=
LFALGS=

${TARGET}: ${OBJFILES}
   ${CC} -o ${TARGET} ${OBJFILES}
clean:
   rm -f *.o ${TARGET}
dist: ${DISTFILES}
   tar cvf ${TARGET}_v${VERSION}.tar ${DISTFILES}
   gzip ${TARGET}_v${VERSION}.tar

.c.o:
   ${CC} ${CFLAGS} ${LFLAGS} -c $*.c
```

tlbe.c

```
/* Tlearn bit error (testing) calculation
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG August 1999
 * Date: 18th August, 1999
 * Modified: 9th November, 1999
 * Version: 0.24
 */

/* --- Includes --- */

#include <stdio.h>
#include <string.h>
#include <unistd.h>     /* for exit() */
#include <stdlib.h>
#include "tokenlst.h"
#include "tokscan.h"
#include "stddefs.h"


/* --- Defines --- */

#define VERSION   "0.24"
```

```
#define HELPFILE  "tlbe.help"

#define  APPNAME     0
#define  TAGSFILE    1
#define  INPUTFILE   2
#define  PATTERNFILE 3
#define  OUTPUTFILE  4
#define  NUMARGS     5

#define  MAXTAGLENGTH   10


/* --- Helper functions' definitions --- */

void Usage(FILE *output, char *appName);
void ProcessCommandLine(int argc, char *argv[], FILE **tagsFile, FILE **inputFile, FILE **
patternFile, FILE **outputFile);
void Abort(int retnum);
void Help(char *appName);
void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs);
tokenList *ReadWholeFile(FILE *input);
void ReadPattern(float patternArray[], FILE *inputFile, bool *inFile);
double OutputSum(float patternArray[], int numOutputs);
int ReadTarget(FILE *patternFile, bool *inFile);
int MaxSlot(float patternArray[], int numOutputs);
void CloseFiles(FILE *tagsFile, FILE *inputFile, FILE *patternFile, FILE *outputFile);
bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize);


/* --- Main --- */

int main(int argc, char *argv[])
{
   FILE     *tagsFile, *inputFile, *patternFile, *outputFile;
   char     **tagsArray;
   float    *patternArray;
   int      numPatterns, numOutputs, target, predicted;
   long     bitError;
   double   avOutputSum;
   bool     inFile;

   ProcessCommandLine(argc, argv,
            &tagsFile, &inputFile, &patternFile, &outputFile);

   ReadTags(tagsFile, &tagsArray, &numOutputs);

   if ((patternArray = (float *) malloc(sizeof(float) * numOutputs)) == NULL) {
      fprintf(stderr, "*** Could not allocate pattern array.\n");
      Abort(7);
   }

   printf(" - TlearnBitError ver %s\n", VERSION);
   printf(" | Input file (tlearn output sweeps) [%s]\n", argv[INPUTFILE]);
   printf(" | Pattern file (targets) [%s]\n", argv[PATTERNFILE]);
   printf(" +-> Output file [%s", argv[OUTPUTFILE]);

   outputFile == stdout ? printf(" (stdout)]\n\n") : printf("]\n\n");

   printf("Read %d tags (%d output lines)\n", numOutputs, numOutputs);

   numPatterns = 0;
   avOutputSum = 0.0;
   bitError = 0;
   inFile = TRUE;
   while (inFile) {
      ReadPattern(patternArray, inputFile, &inFile);
      avOutputSum += OutputSum(patternArray, numOutputs);

      numPatterns++;

      target = ReadTarget(patternFile, &inFile);
      predicted = MaxSlot(patternArray, numOutputs);

      if (target != predicted)
         bitError++;

      if (numPatterns > 100 && (numPatterns % 1000 == 0))
         fprintf(stderr, "Processed %d patterns...\n", numPatterns);
   }
   fprintf(stderr, "Processed %d patterns total.\n", numPatterns - 1);
```

```
   fprintf(outputFile, "%d\t%5.2f%%\t%5.2f%%\t%.2f\n", bitError, bitError / (float) numPatterns *
100.0, (1.0 - (bitError / (float) numPatterns)) * 100.0, avOutputSum / (float) numPatterns);

   CloseFiles(tagsFile, inputFile, patternFile, outputFile);

   return 0;
}


/* --- Helper functions --- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** TlearnBitError ver %s -- outputs learning error statistics from tlearn\n",
VERSION);
   fprintf(output, "Usage: %s {tagsFile} {inputFile} {patternFile} {outputFile}\n", appName);
   fprintf(output, "Where: {tagsFile}    -- tags representing input lines\n");
   fprintf(output, "       {inputFile}   -- output from tlearn verify sweep\n");
   fprintf(output, "                        (specify '-' for stdin)\n");
   fprintf(output, "       {patternFile} -- file to specify output targets\n");
   fprintf(output, "       {outputFile}  -- file to append output to\n");
   fprintf(output, "                           format: (#errors) (av. bit error) (av. correct) (av.
output sum)\n");
   fprintf(output, "                        (specify '-' for stdout)\n");
   fprintf(output, "For detailed info, type %s --help\n\n", appName);
}

void ProcessCommandLine(int argc, char *argv[], FILE **tagsFile, FILE **inputFile, FILE
**patternFile, FILE **outputFile)
{
   if ((!(argc < 2)) && ((strcasecmp(argv[1], "--help") == 0))) {
      Help(argv[APPNAME]);
      exit(0);
   }

   if (argc < NUMARGS) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      Abort(1);
   }

   if (argc > NUMARGS)
      fprintf(stderr, "--- Warning: extra arguments ignored.\n");

   if ((*tagsFile = fopen(argv[TAGSFILE], "rt")) == NULL) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Could not open tags file [%s].\n", argv[TAGSFILE]);
      Abort(3);
   }

   if (strcasecmp(argv[INPUTFILE], "-") == 0)
      *inputFile = stdin;
   else if ((*inputFile = fopen(argv[INPUTFILE], "rt")) == NULL) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Could not open input file [%s].\n", argv[INPUTFILE]);
      Abort(4);
   }

   if ((*patternFile = fopen(argv[PATTERNFILE], "rt")) == NULL) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Could not open pattern file [%s].\n", argv[PATTERNFILE]);
      Abort(5);
   }

   if (strcasecmp(argv[OUTPUTFILE], "-") == 0)
      *outputFile = stdout;
   else if ((*outputFile = fopen(argv[OUTPUTFILE], "at")) == NULL) {
      Usage(stderr, argv[APPNAME]);
      fprintf(stderr, "*** Could not open output file [%s].\n", argv[OUTPUTFILE]);
   }
}

void Abort(int retnum)
{
   fprintf(stderr, "Aborting...\n");
   exit(retnum);
}

void Help(char *appName)
{
   FILE *helpFile;
```

```
   if ((helpFile = fopen(HELPFILE, "wt")) == NULL) {
      fprintf(stderr, "*** Could not create help file [%s].\n", HELPFILE);
      Abort(2);
   }

   fprintf(helpFile, "-----------------------------------------\n");
   fprintf(helpFile, "   Help file for TlearnBitErr ver %s\n", VERSION);
   fprintf(helpFile, "-----------------------------------------\n\n");

   Usage(helpFile, appName);

   fprintf(helpFile, "\nThis program is used to output a table of stats derived\n");
   fprintf(helpFile, "from an Elman Tlearn testing (verification) sweep.\n");
   fprintf(helpFile, "It takes as input the following files:\n");
   fprintf(helpFile, "   o Tags file: Contains signatures for each ouput line,\n");
   fprintf(helpFile, "     separated by whitespace.\n");
   fprintf(helpFile, "   o Input file: (stdin if field = '-') Direct output from\n");
   fprintf(helpFile, "     tlearn verify option.\n\n");

   fprintf(helpFile, "It appends to the output file the averaged bit error and the\n");
   fprintf(helpFile, "averaged sum of the outputs.\n");

   fprintf(stderr, "Created %s\n", HELPFILE);
}


void ReadTags(FILE *tagsFile, char **tagsArray[], int *numOutputs)
{
   tokenList   *allTags, *index;
   int      numTags;

   allTags = ReadWholeFile(tagsFile);

   if (allTags == NULL) {
      fprintf(stderr, "*** Error reading token file.\n");
      Abort(5);
   }

   numTags = 0;
   index = allTags;
   while (index != NULL) {
      index = index -> NEXT;
      numTags++;
   }

   *numOutputs = numTags;

   if (!Allocate2DArray(tagsArray, numTags, 1, MAXTAGLENGTH)) {
      fprintf(stderr, "*** Could not allocate tag array.\n");
      Abort(6);
   }

   index = allTags;
   numTags = 0;
   while (index != NULL) {
      strcpy((*tagsArray)[numTags], index -> token);
      index = index -> NEXT;
   }

   DestroyTokenList(allTags);
}

tokenList *ReadWholeFile(FILE *input)
{
   tokenList   *list, *temp;
   bool     inFile;

   inFile = TRUE;
   list = ReadLineTokens(input, &inFile);    /* Get the first line */

   if (list == NULL)
      return list;

   while (inFile) {
      temp = ReadLineTokens(input, &inFile); /* Get the next line */
      list = ConcatenateTokenList(list, temp);
   }

   return list;
}

void ReadPattern(float patternArray[], FILE *inputFile, bool *inFile)
```

```
{
   tokenList   *outputs, *outputsIndex;
   int      arrayIndex;

   if ((outputs = ReadLineTokens(inputFile, inFile)) == NULL) {
      if (!*inFile) return;
      fprintf(stderr, "*** Error reading input file.\n");
      Abort(8);
   }

   arrayIndex = 0;
   outputsIndex = outputs;
   while (outputsIndex != NULL) {
      patternArray[arrayIndex] = atof(outputsIndex -> token);
      outputsIndex = outputsIndex -> NEXT;
      arrayIndex++;
   }

   DestroyTokenList(outputs);
}

double OutputSum(float patternArray[], int numOutputs)
{
   double   sum = 0;

   while (numOutputs > 0)
      sum += patternArray[--numOutputs];

   return sum;
}

int ReadTarget(FILE *patternFile, bool *inFile)
{
   tokenList   *pattern, *tIndex;
   int      target;

   if (!(pattern = ReadLineTokens(patternFile, inFile)))
      return 0;

   tIndex = pattern;
   while (tIndex -> NEXT != NULL)
      tIndex = tIndex -> NEXT;

   target = atoi(tIndex -> token);
   pattern = DestroyTokenList(pattern);

   return target;
}

int MaxSlot(float patternArray[], int numOutputs)
{
   int   maxSlot;
   float max;

   max = 0;
   maxSlot = 0;

   while (numOutputs > 0) {
      if (patternArray[numOutputs - 1] > max) {
         maxSlot = numOutputs;
         max = patternArray[numOutputs - 1];
      }
      numOutputs--;
   }
   return maxSlot;
}

void CloseFiles(FILE *tagsFile, FILE *inputFile, FILE *patternFile, FILE *outputFile)
{
   fclose(tagsFile);
   fclose(inputFile);
   fclose(patternFile);
   fclose(outputFile);
}

bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize)
{
   int   index;

   if ((*array = malloc(xSize * sizeof(void *))) == NULL)
      return FALSE;
```

```
   for (index = 0; index < xSize; index++) {
      if (((*array)[index] = malloc(ySize * elementSize)) == NULL)
         printf("ySize\n");

      xSize--;
   }
   return TRUE;
}

/* --- END of tlbe.c --- */
```

9.2.9 sclust

Application purpose

sclust performs adaptive spherical cluster analysis on a data set. The data can be of any dimensionality. sclust uses a modified adaptive Forgy's algorithm, and is deterministic (i.e. the analysis only needs to be performed once, and will always return the best result for the algorithm used). See section 1.0 for a description of cluster analysis and section 3.1 for a description of the adaptive algorithm.

Usage

```
*** sclust -- Performs adaptive clustering on a set of vectors
    Version 0.17 build build 17:01:17 Jun 30 2000
Usage: sclust {vector file} {means file} {av. dist. parm}
Where: {vector file}    -- file of vectors to analyse, one per line
                           ['-' for stdin]
       {means file}     -- filename to output the mean vectors to
                           ['-' for stdout]
       {av. dist. parm} -- see help file for information
                           (0.5 is a good value)
```

Modules used

StdDefs, TokenLst, TokScan, Vector_Utils, Vector_Read, Cluster

Source code

Makefile

```
TARGET=sclust
VERSION=017
CFILES=tokenlst.c tokscan.c vector_utils.c vector_read.c sclust.c cluster.c
HFILES=tokenlst.h tokscan.h vector_utils.h vector_read.h stddefs.h cluster.h
OBJFILES=tokenlst.o tokscan.o vector_utils.o vector_read.o sclust.o cluster.o
DISTFILES=Makefile sclust.help.make ${CFILES} ${HFILES} ${TARGET}

CC=cc
CFLAGS=-O2 -64 -apo
LFLAGS=-lm

${TARGET}: ${OBJFILES}
   ${CC} ${CFLAGS} -o ${TARGET} ${OBJFILES} ${LFLAGS}

help: ${TARGET}
   ${TARGET} --help

clean:
   rm -f *.o ${TARGET}

refresh: clean ${TARGET}

dist: ${DISTFILES}
   tar cvf ${TARGET}_v${VERSION}.tar ${DISTFILES}
   gzip -f ${TARGET}_v${VERSION}.tar

.c.o:
   ${CC} ${CFLAGS} -c $*.c
```

sclust.help.make

```
/* Help make file for sclust
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 * Date: 16th September, 1999
 * Modified:
 * Version: 0.02
 */

#define p(x)   fprintf(helpFile, x);

p("This program performs adaptive clustering (a modified Forgy'sAlgorithm)\n");
p("on a set of vectors.  This is a data compression technique also known as\n");
```

```
p("Vector Quantisation.\n");
p("\n");
p("Inputs are:\n");
p(" o A file of vectors, all of the same dimensionality, to perform clustering\n");
p("   on (the training set).\n");
p(" o A parameter that determines the distance ratio required for a pattern to\n");
p("   be used as the seed of a new cluster.\n");
p("\n");
p("Input formats:\n");
p("Vector files (training set, seeds):\n");
p(" o Vectors are input one per line\n");
p(" o Vector components are separated by whitespace (space or tab)\n");
p(" o All vectors in a file should be of the same dimensionality\n");
p("\n");
p("Output formats:\n");
p("Vector files (means output)\n");
p(" o Vectors are output one per line\n");
p(" o Vector components are tab-delimited\n");
p("\n");
p("Note that a '-' can be supplied on the command line to denote reading from\n");
p("stdin and writing to stdout.\n");
p("\n");
p("\n");

#undef p

/* --- END of sclust.help.make --- */
```

sclust.c

```
/* sclust -- performs vector quantization on a set of vectors
 *            uses an adaptive modified Forgy's Algorithm to find
 *            the means of a set of clusters that represent the
 *            entire set
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 15th September, 1999
 * Modified: 21st November, 2000
 * Version: 0.17
 *
 * Usage: sclust {input vectors file}  -- whitespace-delimited, one vector per line
 *                                        (accepts stdin if '-' is specified)
 *               {output means file}   -- tab-delimited, one mean vector per line
 *                                        (writes to stdout if '-' is specified)
 *               {av. dist. parameter} -- average distance a pattern must be from
 *                                        all clusters to create a new cluster
 *
 * Modules used: vector_utils, vector_read, cluster, stddefs, tokenlst, tokscan
 *
 * Aknowledgements: Towsey, M. 1998, "The Use of Neural Networks in the Automated Analysis
 *                   of the Electroencephalogram", Phd. Thesis, Univ. of QLD, Brisbane.
 *                  Sayood, K. 1996, "Introduction to Data Compression", Morgan Kaufmann
 *                   Publishers, San Francisco.
 *                  Das, S. & Moser, M. 'Dynamic On-Line Clustering and State Extraction:
 *                   An Approach to Symbolic Learning', "Neural Networks", vol. 11, no. 1, pp.53-64.
 *
 * Note: This code accompanies the report "To Build a Better Cluster"
 */

/* -- Required modules -- */

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>
#if defined(UNIX)
   #include <unistd.h>
#endif
#if defined(WIN32)
   #include <string.h>
   #define  strcasecmp  _stricmp
#endif
#include "vector_utils.h"
#include "vector_read.h"
#include "cluster.h"
#include "stddefs.h"
```

```
/* -- kmeans defines -- */

#define  APP_NAME    0
#define  VECTOR_FILE 1
#define  HELP        1
#define  MEANS_FILE  2
#define  DIST_PARM   3
#define  NUMARGS     4

#define  VERSION  "0.17"

#define  MAX_CLUSTERS   5000

#define  Abort(x)    AbortPrint((x), __LINE__, __FILE__);
#define  SQR(x)      ((x) * (x))
#define  ABS(x)      ((x) < 0 ? (-(x)) : (x))

/* -- typedefs -- */

typedef  vector   **codebook;    /* an array of vector pointers */


/* -- helper function prototypes -- */

/* -- Function Usage
 * Purpose: Writes the program usage to 'output'
 */
void Usage(FILE *output, char *appName);

/* -- Function Help
 * Purpose: Writes the program help to 'output'
 */
void Help(FILE *output, char *appName);

/* -- Function AbortPrint
 * Purpose: Abnormal termination of the program.
 *          Writes the filename and line number, and terminates
 *          with error code 'code'
 */
void AbortPrint(int code, int line, char *file);

/* -- Function ProcessCommandLine
 * Inputs: 'argc', 'argv' : Command line arguments
 * Purpose: Retrieves the command line parameters
 * Outputs: 'vectorFile' - the file to read training patterns from
 *          'meansFile'  - the file to write the cluster centroids to
 *          'distParm'   - the distance parameter to use for clustering
 */
void  ProcessCommandLine(int  argc,  char  *argv[],  FILE  **vectorFile,  FILE  **meansFile,  float
*distParm);

/* -- Function CloseFiles
 * Pre: 'vectorFile' and 'meansFile' have been opened
 * Post: 'vectorFile' and 'meansFile' have been closed
 */
void CloseFiles(FILE *vectorFile, FILE *meansFile);

/* -- Function PrintBanner
 * Pre: all inputs have been previously initialised
 * Purpose: Prints a program benner describing the clustering to
 *          perform to stderr
 */
void PrintBanner(char *argv[], FILE *vectorFile, FILE *meansFile, float distParm);

/* -- Function ReadTrainingSet
 * Pre: 'vectorFile' has been opened for reading
 *      'numVectors' has been allocated
 * Post: a cluster was returned containing the training data
 *       && 'numVectors' contains the number of patterns read
 */
cluster *ReadTrainingSet(FILE *vectorFile, int *numVectors);

/* -- Function MakeClusters
 * Purpose: Creates 'numClusters' clusters
 *          Returns an array of clusters [0..numClusters-1]
 */
cluster **MakeClusters(int numClusters);

/* -- Function DestroyClusters
 * Purpose: Destroys a cluster array[0..numClusters-1]
 *          Returns NULL
 */
```

```
cluster **DestroyClusters(cluster **clusters, int numClusters);

/* -- Function FindNearestNeighbour
 * Pre: 'vect' is the data pattern to test
 *      'currentBook' is the current codebook of cluster centroids
 *      'numClusters' is the number of valid clusters in 'currentBook'
 * Post: Using the Euclidean distance, returns the entry in
 *       'currentBook' that is closest to 'vect'
 * Note: If there are two closest clusters, the function returns the first
 */
int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters);

/* -- Function FindFurthestVector
 * Pre: 'vectorSet' is the array of all training vectors
 *      'numTrainVectors' is the number of entries in 'vectorSet'
 *      'clusterMembership' is the array associating training vectors with clusters
 *      'currentBook' is the current codebook
 * Post: Using Euclidean distance, returns the index into 'vectorSet' that is
 *       furthest away from its classified cluster
 */
int FindFurthestVector(codebook vectorSet, int numTrainVectors, int clusterMembership[], codebook
currentBook);

/* -- Function GetError
 * Purpose: Calculates the classification error of the current codebook
 */
double GetError(cluster **clusters, codebook book, int numClusters);

/* -- Function ClusterRMSDistance
 * Purpose: Computes the mean square average distance for the patterns
 *          within a cluster
 */
double ClusterRMSDistance(cluster *clust, vector *vect);

/* -- Function InitialiseCodebook
 * Purpose: Sets up the inital codebook for the modified Forgy's
 *          algorithm.  See the text for details
 */
codebook InitialiseCodebook(int numClusters, cluster *vectorSet, int numSlots);

/* -- Function CodebookAverageDistance
 * Purpose: Computes the average distance for a vectors to all other clusters
 */
double CodebookAverageDistance(vector *vect, codebook book, int numClusters);

/* -- Function CodebookCentroid
 * Purpose: Computes the centroid (Euclidean mean) of a cluster
 *          Returns the centroid as a vector
 */
vector *CodebookCentroid(codebook book, int numSlots);

/* -- Function FreeCluster
 * Purpose: Finds the first free cluster slot in the cluster array
 */
int FreeCluster(codebook book, int numSlots);


/* -- Codebook functions -- */

/* -- Function CodebookNew
 * Purpose: Creates a new codebook with 'slots' entries
 */
codebook CodebookNew(int slots);

/* -- Function CodebookDestroy
 * Purpose: Destroys a codebook with 'slots' entries
 */
codebook CodebookDestroy(codebook book, int slots);

/* -- Function CodebookNewFromCluster
 * Pre:'clust' is an allocated cluster
 * Post: A codebook was created containing an entry for each
 *       vector in 'clust'
 */
codebook CodebookNewFromCluster(cluster *clust);

/* -- Function CodebookMakeCopy
 * Purpose: Creates a complete copy of a codebook
 */
codebook CodebookMakeCopy(codebook book, int slots);

/* -- Function WriteCodebook
```

```
 * Purpose: Writes the vectors in 'book' to 'output'
 */
void WriteCodebook(FILE *output, codebook book, int slots);


/* -- main -- */

int main(int argc, char *argv[])
{
   FILE     *vectorFile,    /* The file to read the training patterns from */
            *meansFile;     /* The file to write the cluster centroids to */
   int      lastCluster;    /* The last valid entry in the cluster array */
   bool     terminate;      /* flag indicating the completion of clustering */
   float    distParm;       /* param.dist, as described in the text */
   double   lastError,      /* Previous calssification error */
            thisError;      /* Current classification error */

   codebook currentBook,    /* The current set of centroids */
            vectorSet;      /* Set of all training vectors */

   int   slot,                 /* An index into various arrays */
         oldClosest,           /* Previsous closest cluster for a vector */
         numTrainVectors,      /* How many vectors read from traing file */
         *clusterMembership;   /* What cluster does this vector belong to? */

   cluster  **clusters,           /* Array of current clusters */
            *vectorSetCluster;    /* Contains the entire training set */

   double   avDist,         /* The average distance from all vectors to their closest cluster */
            closestDist;    /* The distance to the closest cluster */
   int      closest;        /* The index of the closest cluster in 'clusters' */


   /* Read in the program parameters */
   ProcessCommandLine(  argc, argv,
                        &vectorFile, &meansFile, &distParm);

   /* Display the function to perform */
   PrintBanner(argv, vectorFile, meansFile, distParm);

   /* Input training set */
   vectorSetCluster = ReadTrainingSet(vectorFile, &numTrainVectors);
   vectorSet = CodebookNewFromCluster(vectorSetCluster);


   /* Initialise cluster membership array */
   if (!(clusterMembership = (int *) malloc(numTrainVectors * sizeof(int)))) {
      fprintf(stderr, "*** Could not allocate cluster membership array.\n");
      Abort(24);
   }

   /* Initialise codebook */
   {
      int   furthestVect;     /* Index of furthest vector */

      for (slot = 0; slot < numTrainVectors; slot++)
         clusterMembership[slot] = 0;

      currentBook = InitialiseCodebook(MAX_CLUSTERS, vectorSetCluster, numTrainVectors);
      furthestVect = FindFurthestVector(vectorSet, numTrainVectors, clusterMembership, currentBook);
      currentBook[1] = VectorMakeCopy(vectorSet[furthestVect]);

      /* Initialise clusters array */
      clusters = MakeClusters(MAX_CLUSTERS);
      clusters[0] = ClusterMakeCopy(vectorSetCluster);
      clusters[0] = ClusterRemoveVector(clusters[0], vectorSet[furthestVect]);
      clusters[1] = ClusterAddVector(clusters[1], VectorMakeCopy(vectorSet[furthestVect]));

      lastCluster = 1;
   }

   /* Get the initial training error */
   lastError = GetError(clusters, currentBook, lastCluster + 1);

   terminate = FALSE;
   while (!terminate) {
      slot = 0;
      while (slot < numTrainVectors) {
         avDist = CodebookAverageDistance(vectorSet[slot], currentBook, lastCluster + 1);
         closest = FindNearestNeighbour(vectorSet[slot], currentBook, lastCluster + 1);
         closestDist = VectorEuclideanDist(vectorSet[slot], currentBook[closest]);
```

```
         if (closestDist > (avDist * distParm)) {
            /* Make a new cluster with the current vector */
            closest = FreeCluster(currentBook, MAX_CLUSTERS);
            if (closest == -1) {
               fprintf(stderr, "*** Exceeded compiler limit of clusters [%d].\n", MAX_CLUSTERS);
               Abort(22);
            }
            if (closest > lastCluster)
               lastCluster = closest;
         }

         oldClosest = clusterMembership[slot];

         /* Update the previous cluster's centroid after removing the vector */
         if (oldClosest != -1) {
            clusters[oldClosest] = ClusterRemoveVector(clusters[oldClosest], vectorSet[slot]);
            if (currentBook[oldClosest] != NULL) VectorDeallocate(currentBook[oldClosest]);
            currentBook[oldClosest] = ClusterCentroid(clusters[oldClosest]);
         }

         /* Update the current cluster's centroid after adding the vector */
         clusters[closest] = ClusterAddVector(clusters[closest], VectorMakeCopy(vectorSet[slot]));
         clusterMembership[slot] = closest;

         if (currentBook[closest] != NULL) VectorDeallocate(currentBook[closest]);
         currentBook[closest] = ClusterCentroid(clusters[closest]);

         slot++;     /* Next training vector */
      }

      /* Get current training error */
      thisError = GetError(clusters, currentBook, lastCluster + 1);
      if ((lastError >= thisError) && (lastError - thisError < 0.0001))
         terminate = TRUE;

      fprintf(stderr, "RMS error: %f (%d clusters)\n", thisError, lastCluster + 1);
      lastError = thisError;
   }

   /* Output final clusters */
   WriteCodebook(meansFile, currentBook, MAX_CLUSTERS);

   /* Clean up and exit */
   clusters = DestroyClusters(clusters, MAX_CLUSTERS);

   CloseFiles(vectorFile, meansFile);

   return 0;
}


/* -- helper functions -- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** sclust -- Performs adaptive clustering on a set of vectors\n");
   fprintf(output, "    Version %s build %s %s\n", VERSION, __TIME__, __DATE__);
   fprintf(output, "Usage: %s {vector file} {means file} {av. dist. parm}\n", appName);
   fprintf(output, "Where: {vector file}    -- file of vectors to analyse, one per line\n");
   fprintf(output, "                           ['-' for stdin]\n");
   fprintf(output, "       {means file}     -- filename to output the mean vectors to\n");
   fprintf(output, "                           ['-' for stdout]\n");
   fprintf(output, "       {av. dist. parm} -- see help file for information\n");
   fprintf(output, "                           (0.5 is a good value)\n");
   fprintf(output, "For detailed info, type %s --help\n", appName);
   fprintf(output, "\n");
 }

void AbortPrint(int code, int line, char *file)
{
   fprintf(stderr, "Aborting [%s | %d]...\n", file, line);
   exit(code);
}


void ProcessCommandLine(int argc, char *argv[], FILE **vectorFile, FILE **meansFile, float
*distParm)
{
   if ((argc > 1) && (strcasecmp(argv[HELP], "--help") == 0)) {
      /* Write help to a file */
      FILE  *helpFile;
```

```
      if (!(helpFile = fopen("sclust.help", "wt"))) {
         fprintf(stderr, "*** Could not create help file [sclust.help].\n");
         Abort(15);
      }

      Help(helpFile, argv[APP_NAME]);
      printf("Created sclust.help\n");
      exit(0);
   }

   if (argc < NUMARGS) {
      /* Not enough arguments */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      Abort(1);
   }

   if (argc > NUMARGS) {
      /* Too many arguments */
      fprintf(stderr, "--- Warning: extra arguments ignored.\n");
   }

   if (strcmp(argv[VECTOR_FILE], "-") == 0) {
      /* Read training vectors from console */
      *vectorFile = stdin;
   }

   else if (!(*vectorFile = fopen(argv[VECTOR_FILE], "rt"))) {
      /* Could not open file to read training vectors from */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open vector file [%s] for reading.\n", argv[VECTOR_FILE]);
      Abort(2);
   }

   if (strcmp(argv[MEANS_FILE], "-") == 0) {
      /* Write training vectors to console */
      *meansFile = stdout;
   }

   else if (!(*meansFile = fopen(argv[MEANS_FILE], "wt"))) {
      /* Could not open file to write clusters to */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open means file [%s] for writing.\n", argv[MEANS_FILE]);
      Abort(3);
   }

   if (!isdigit(*argv[DIST_PARM])) {
      /* Could not translate distance parameter */
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Error converting distance parameter from [%s].\n", argv[DIST_PARM]);
      Abort(5);
   }

   *distParm = (float) atof(argv[DIST_PARM]);
   if ((*distParm > 2.0) || (*distParm < 0.0)) {
      /* Invalid distance parameter */
      fprintf(stderr, "*** Distance parameter [%2f] must be between 0.0 and 1.0.\n");
      Abort(26);
   }
}

void CloseFiles(FILE *vectorFile, FILE *meansFile)
{
   if (vectorFile != stdin) fclose(vectorFile);
   if (meansFile != stdout) fclose(meansFile);
}


void PrintBanner(char *argv[], FILE *vectorFile, FILE *meansFile, float distParm)
{
   fprintf(stderr, "- sclust ver %s\n", VERSION);
   fprintf(stderr, " o Vectors: << [");
   vectorFile == stdin ?
      fprintf(stderr, "- stdin]\n") :
      fprintf(stderr, "%s]\n", argv[VECTOR_FILE]);

   fprintf(stderr, " o Means: >> [");
   meansFile == stdout ?
      fprintf(stderr, "- stdout]\n") :
      fprintf(stderr, "%s]\n", argv[MEANS_FILE]);

   fprintf(stderr, " o distance parameter: [%f]\n", distParm);
```

```
}


codebook CodebookNew(int slots)
{
   codebook newBook;

   if (!(newBook = (vector **) malloc(slots * sizeof(vector *))))
      return NULL;

   return newBook;
}

codebook CodebookNewFromCluster(cluster *clust)
{
   codebook newBook;    /* new codebook */
   cluster  *cIndex;    /* index into cluster */
   int      slot;       /* slot index into codebook */

   if (!(newBook = CodebookNew(ClusterSize(clust))))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      /* Copy each vector into the new codebook */
      newBook[slot] = VectorMakeCopy(cIndex -> vect);
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

codebook CodebookDestroy(codebook book, int slots)
{
   int   slot;

   slot = 0;
   while (slot < slots) {
      VectorDeallocate(book[slot]);
      slot++;
   }

   free(book);
   return NULL;
}

cluster *ReadTrainingSet(FILE *vectorFile, int *numVectors)
{
   cluster     *vectorSet;

   fprintf(stderr, "Reading training vectors...\n");

   /* Read in vectors as a cluster */
   vectorSet = ClusterReadFromFile(vectorFile);
   if (ClusterIsEmpty(vectorSet)) {
      fprintf(stderr, "*** Empty training set.\n");
      Abort(10);
   }

   /* Get the cluster size */
   *numVectors = ClusterSize(vectorSet);
   fprintf(stderr, "Read %d vectors.\n", *numVectors);

   if (!ClusterCheckDimensionality(vectorSet)) {    /* make sure they're all the same dimensionality
*/
      fprintf(stderr, "*** The training vectors must all be the same dimensionality.\n");
      Abort(7);
   }
   fprintf(stderr, "Training dimension: %d\n", vectorSet -> vect -> dimensions);

   return vectorSet;
}

codebook CodebookMakeCopy(codebook book, int slots)
{
   codebook newBook;
   int      slot;       /* index into codebook slots */

   if (!(newBook = CodebookNew(slots)))
      return NULL;
```

```
   slot = 0;
   while (slot < slots) {
      /* Copy each vector into the new codebook */
      newBook[slot] = VectorMakeCopy(book[slot]);
      slot++;
   }

   return newBook;
}

cluster **MakeClusters(int numSeeds)
{
   cluster  **newClusterArray;
   int      slot;       /* index into cluster array */

   if (!(newClusterArray = (cluster **) malloc(numSeeds * sizeof(cluster *)))) {
      fprintf(stderr, "*** Unable to allocate cluster space.\n");
      Abort(14);
   }

   slot = 0;
   while (slot < numSeeds) {
      /* Allocate each cluster */
      newClusterArray[slot] = ClusterNew();
      slot++;
   }

   return newClusterArray;
}

cluster **DestroyClusters(cluster **clusters, int numSeeds)
{
   int   slot;

   slot = 0;
   while (slot < numSeeds) {
      clusters[slot] = ClusterDestroy(clusters[slot]);
      slot++;
   }

   free (clusters);

   return NULL;
}


int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters)
{
   int      slot,    /* index into codebook */
            minSlot; /* current slot of minimum distance */
   double   minDist, /* current minimum distance */
            dist;    /* current distance */

   minSlot = 0;
   minDist = VectorEuclideanDist(vect, currentBook[0]);
   slot = 1;
   while (slot < numClusters) {
      if (currentBook[slot] != NULL) {
         dist = VectorEuclideanDist(vect, currentBook[slot]);

         if (dist < minDist) {
            minDist = dist;
            minSlot = slot;
         }
      }
      slot++;
   }

   return minSlot;
}

double GetError(cluster **clusters, codebook book, int numClusters)
{
   double   bookError;
   int   slot;

   bookError = 0.0;
   slot = 0;
   while (slot < numClusters) {
      bookError += ClusterRMSDistance(clusters[slot], book[slot]);
      slot++;
```

```
   }

   bookError /= (double) numClusters;
   return sqrt(bookError);
}

double ClusterRMSDistance(cluster *clust, vector *vect)
{
   double   errSum;
   cluster  *cIndex;

   if (ClusterIsEmpty(clust))
      return 0.0;

   errSum = 0.0;
   cIndex = clust;
   while (cIndex != NULL) {
      errSum += SQR(VectorEuclideanDist(vect, cIndex -> vect));
      cIndex = cIndex -> next;
   }

   errSum /= (double) ClusterSize(clust);
   return sqrt(errSum);
}

void WriteCodebook(FILE *output, codebook book, int slots)
{
   int   slot;

   for (slot = 0; slot < slots; slot++) {
      if (book[slot] != NULL) {
         VectorWrite(output, book[slot]);
         fprintf(output, "\n");
      }
   }
}

void Help(FILE *helpFile, char *appName)
{
   fprintf(helpFile, "------------------------------------\n");
   fprintf(helpFile, "   Help file for sclust ver %s\n", VERSION);
   fprintf(helpFile, "------------------------------------\n\n");

   Usage(helpFile, appName);

#include "sclust.help.make"
}

codebook InitialiseCodebook(int numClusters, cluster *vectorSet, int numSlots)
{
   codebook newBook;
   int      slot;

   if (!(newBook = CodebookNew(numClusters))) {
      fprintf(stderr, "*** Could not initialise codebok.\n");
      Abort(20);
   }

   if (!(newBook[0] = ClusterCentroid(vectorSet))) {
      fprintf(stderr, "*** Could not create initial centroid vector.\n");
      Abort(21);
   }

   for (slot = 1; slot < numClusters; slot++)
      newBook[slot] = NULL;

   return newBook;
}

vector *CodebookCentroid(codebook book, int numSlots)
{
   vector   *vectSum;      /* Current sum of vectors in codebook */
   int      slot, entries;

   if (numSlots == 0)
      return NULL;

   if (!(vectSum = VectorZero(book[0] -> dimensions)))
      return NULL;

   entries = 0;
   slot = 0;
```

```
   while (slot < numSlots) {
      if (book[slot] != NULL) {
         VectorSum(vectSum, book[slot]);
         entries++;
      }
      slot++;
   }

   /* Find the average by dividing by the number of vectors */
   VectorDivideScalar(vectSum, (double) entries);
   return vectSum;
}

int FreeCluster(codebook book, int numSlots)
{
   int   slot;

   for (slot = 0; slot < numSlots; slot++)
      if (book[slot] == NULL)
         return slot;

   return -1;
}


double CodebookAverageDistance(vector *vect, codebook book, int numClusters)
{
   double  distance;    /* Cumulative distance */
   int     slot;        /* Index into codebook */

   distance = 0.0;
   slot = 0;
   while (slot < numClusters) {
      if (book[slot] == NULL) {
         numClusters--;
         slot++;
         continue;
      }
      distance += VectorEuclideanDist(vect, book[slot]);
      slot++;
   }

   return distance / (double) numClusters;
}


int FindFurthestVector(codebook vectorSet, int numTrainVectors, int clusterMembership[], codebook
currentBook)
{
   double   distance,      /* Current test distance */
            furthestDist;  /* Current furthest distance */
   int      slot,          /* index into 'vectorSet' */
            furthestSlot;  /* Current furthest vector */

   slot = 0;
   furthestSlot = -1;
   furthestDist = 0.0;

   while (slot < numTrainVectors) {
      distance = VectorEuclideanDist(vectorSet[slot], currentBook[clusterMembership[slot]]);

      if (distance > furthestDist || furthestSlot == -1) {
         furthestDist = distance;
         furthestSlot = slot;
      }

      slot++;
   }

   return furthestSlot;
}

/* --- END of sclust.c --- */
```

9.2.10 o2clust

Application purpose

o2clust performs cluster analysis on a set of data. It uses an adaptive algorithm outlined in section 3.2 tailored to the second-order distance metric described in section 2.2.

Application notes

Although the correlation-matrix cluster representation implemented in the **corrmatrix** module is complete, the algorithm does not work at present. The problem seems to lie with forming clusters that have too few points, and therefore are unnaturally biased along an arbitrary axis. When forming clusters by adding points (as opposed to splitting larger clusters into progressively smaller clusters) the clusters grow from a few points to encompass (hopefully) a natural cluster. However, when a cluster contains one or two points, the correlation matrix is either singular or very close, and the resulting "shape" of the cluster is merely the axis through the two points. This severe skew persists until a greater number of points are used to form the matrix.

The preliminary solution was to form a "dummy" cluster around a cluster with few points. This dummy cluster would be used to calculate the correlation matrix until a certain threshold number of points existed within the cluster. The dummy cluster was spherical around the centroid of the real cluster. However, this did not solve the problem.

The second approach was to "steal" the nearest *n* points when creating a new cluster. This caused unnaturally shaped clusters.

Lipson and Siegelmann [1998] use the correlation matrix representation to form the receptive field of a neuron. The matrix is incrementally updated (via a "learning rate"). The receptive fields are initialised to an area before learning begins. No new "clusters" or neurons are generated. One approach with o2clust would be to likewise segment the data space to initialise the clusters. This would require some new method to create a new cluster, however.

Another possible method would be to incrementally adjust the correlation matrices via a learning rate so they did not "contain" the points but merely represented a "best guess" of a cluster. The data set could then be classified on the termination of the algorithm.

Usage

```
*** o2clust -- performs adaptive O2 clustering on a set of vectors
    Version 0.32.cp build 22:20:54 Oct 10 2000 [IRIX64 6.5 IP25]
Usage: o2clust {vector file} {cluster file} {classify file} {stop param}
Where: {vector file}   -- file to take vectors from
                          [specify '-' for stdin]
       {cluster file}  -- file to write clusters to
                          [specify '-' for stdout]
       {classify file} -- file to write classified vectors (from vector file) to
                          [specify '-' to skip this step]
       {stop param}    -- stop clustering when RMS error changes by less than
                          this factor.  i.e. 0.1 => 10%%, 0.25 => 25%%, etc.
```

Modules used

StdDefs, 2DArray, TokenLst, TokScan, Gauss, RunAvg, Vector_Utils, Vector_Read, CorrMatrix, Cluster

Source code

Makefile

```
TARGET=o2clust
VERSION=032cp
CFILES=${TARGET}.c  cluster.c  vector_utils.c  vector_read.c  corr_matrix.c  tokscan.c  tokenlst.c
gauss2.c 2darray.c runavg.c
HFILES=stddefs.h cluster.h vector_utils.h vector_read.h corr_matrix.h tokscan.h tokenlst.h gauss.h
runavg.h 2darray.h
OBJFILES=${TARGET}.o  cluster.o  vector_utils.o  vector_read.o  corr_matrix.o  tokscan.o  tokenlst.o
gauss2.o nrutil.o runavg.o 2darray.c
DISTFILES=Makefile o2clust.help.make ${CFILES} ${HFILES} ${TARGET}
SYSVER=\"`uname -mprs`\"

CC=cc
CFLAGS=-64 -pca -DPLATFORM="${SYSVER}" -DUNIX #-DDEBUG
LFLAGS=-64 -pca -lm -lmalloc

${TARGET}: ${OBJFILES}
   ${CC} -o ${TARGET} ${OBJFILES} ${LFLAGS}

help: ${TARGET}
   ${TARGET} --help

clean:
   rm -f *.o ${TARGET}

refresh: clean ${TARGET}

dist: ${DISTFILES}
   tar cvf ${TARGET}_v${VERSION}.tar ${DISTFILES}
   gzip -f ${TARGET}_v${VERSION}.tar

.c.o:
   ${CC} ${CFLAGS} -c $*.c
```

o2clust.help.make

```
/* Help make file for o2clust
 *
 * Author: Dylan Muir
 * Date: 3rd July, 2000
 * Modified:
 * Version: 0.15.cp
 */

#define p(x)   fprintf(helpFile, x);

p("\n");
p("Put your help info here\n");
p("\n");
p("\n");
p("blah blah blah\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");
p("\n");

#undef p

/* --- END of generic.help.make --- */
```

o2clust.c

```
/* o2clust -- performs order-2 (ellipsoid) clustering on a set of vectors
 *            Uses a modified version of the adaptive Forgy's algorithm
 *
 * Author: Dylan Muir(dr.muir@student.qut.edu.au)
 *         QUT MLRC Semester 2, November 1999
 * Date: 8th November, 1999
 * Modified: 10th October, 2000
 * Version: 0.32.cp (cross-platform Win32 and Unix)
 *
```

```
 * Usage: o2clust (vector file) (clusters file) (classified vectors file) (clustering parameter)
 *
 * Modules used: stddefs, vector_utils, vector_read, cluster, corr_matrix, 2darray, gauss2,
 *               runavg
 *
 *  Aknowledgements: Lipson, H., Siegelman, H.T., "High Order Shape Neurons for Data Structure
Decomposition"
 *
 */

/* -- Compilation check -- */

#if !defined(UNIX) && !defined(WIN32)
   #error "Compilation for Unix or Win32 ONLY!"
#endif


/* -- Required modules -- */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#ifdef   UNIX
   #include <unistd.h>
#endif
#ifdef   WIN32
   #include <string.h>
   #define  srand48(x)  srand((unsigned int) (x))
   #define  strcasecmp  stricmp
   #define  PLATFORM    "Generic Win32 System"
#include <conio.h>
#include <ctype.h>
#endif
#include <math.h>
#include "stddefs.h"
#include "cluster.h"
#include "vector_utils.h"
#include "vector_read.h"
#include "corr_matrix.h"
#include "runavg.h"


/* -- defines -- */

#define  APP_NAME       0
#define  HELP           1
#define  VECTOR_FILE    1
#define  CLUSTER_FILE   2
#define  CLASSIFY_FILE  3
#define  CLUST_PARAM    4
#define  NUMARGS        5

#define  MAX_CLUSTERS         3000
#define  INIT_STOP_PARAM      0.001
#define  SAMPLES_TO_AVG       5

#define  VERSION        "0.32.cp"
#define  HELP_FILE      "o2clust.help"

#define  Abort(x) AbortPrint((x), __LINE__, __FILE__);


/* -- codebook stuff -- */

typedef  corrMatrix  *codebook;

codebook CodebookNew(int slots);
codebook CodebookDestroy(codebook book, int slots);
codebook CodebookNewFromCluster(cluster *clust);
codebook CodebookMakeCopy(codebook book, int slots);
void WriteCodebook(FILE *output, codebook book, int slots);
codebook InitialiseCodebook(int numClusters, cluster *vectorSet);
int FreeCluster(codebook book, int numSlots);
void WriteCodebook(FILE *output, codebook book, int slots);
double GetError(codebook book, int numClusters);

/* -- kmeans helper function prototypes -- */

void Usage(FILE *output, char *appName);
void Help(FILE *output, char *appName);
void AbortPrint(int code, int line, char *file);
void ProcessCommandLine(int argc, char *argv[],
```

```
        FILE **vectFile, FILE **clustFile, FILE **classFile,
        double *clustParam);
void CloseFiles(FILE *vectFile, FILE *clustFile, FILE *classFile);
void PrintBanner(char *argv[], FILE *vectFile, FILE *clustFile, FILE *classFile, double clustParam);
cluster *ReadTrainingSet(FILE *vectorFile, int *numTrainVectors);
vector **VectArrayNewFromCluster(cluster *clust);
int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters, bool useCorr, bool
ignore[]);
int  FindWorstVector(vector  **vectorSet,  int  numVectors,  int  clusterMembership[],  codebook
currentBook, bool ignore[]);
int FindWorstVectorInCluster(vector **vectorSet, int numVectors, int clusterMembership[], codebook
currentBook, int clusterSlot, bool useCorr);
double CMatrixAvCorrWithinCluster(corrMatrix matrix);
bool UseCorrelation(codebook currentBook, int numClusters);
void  ClassifyVectors(vector  **vectorSet,  int  numTrainVectors,  int  clusterMembership[],  codebook
currentBook, int lastCluster, bool useCorr, double stopParam, bool ignore[]);
double  AverageCodebookCorrelation(vector  *vect,  codebook  currentBook,  int  numClusters,  double
*stddev);


/* -- main -- */

int main(int argc, char *argv[])
{
   FILE     *vectFile, *clustFile, *classFile;
   double   clustParam,
            lastError, thisError;
   int      lastCluster, numTrainVectors,
            *clusterMembership, slot;
   bool     firstRun, terminate;

   cluster  *vectorSetCluster;
   codebook currentBook;
   vector   **vectorSet;

   double   worstCorr,
            bestCorr,
            testCorr,
            bestDist,
            testDist,
            avCorr,
            stddevCorr;
   int      closest,
            oldClosest,
            nextClosest,
            worst,
            lastNewCluster,
            bestFromClust,
            bestToClust,
            fromClust,
            toClust,
            numClusters;

   bool     *ignore,
            firstPass,
            tracking,
            report;

   ProcessCommandLine(  argc, argv,
                        &vectFile, &clustFile, &classFile,
                        &clustParam);

   PrintBanner(argv, vectFile, clustFile, classFile, clustParam);
   vectorSetCluster = ReadTrainingSet(vectFile, &numTrainVectors);
   vectorSet = VectArrayNewFromCluster(vectorSetCluster);

   if (!(clusterMembership = (int *) malloc(numTrainVectors * sizeof(int)))) {
      fprintf(stderr, "*** Could not allocate cluster membership array.\n");
      Abort(24);
   }

   for (slot = 1; slot < numTrainVectors; slot++)
      clusterMembership[slot] = -1;

   clusterMembership[0] = 0;
   /* clusterMembership[1] = 1; */

   currentBook = InitialiseCodebook(numTrainVectors, vectorSetCluster);
   lastCluster = 1;

   if (!(ignore = malloc(numTrainVectors * sizeof(bool)))) {
      fprintf(stderr, "*** Could not allocate ignore array.\n");
```

```
      Abort(23);
   }

   for (slot = 0; slot < numTrainVectors; slot++)
      ignore[slot] = FALSE;

   lastError = GetError(currentBook, lastCluster + 1);
   printf("Initial (no classification) error: %f\n", lastError);

   if (numTrainVectors > 1000)
      report = TRUE;
   else
      report = FALSE;

   tracking = FALSE;
   terminate = FALSE;
   firstRun = FALSE;
   numClusters = 1;
   while (!terminate) {

      slot = 0;
      while (slot < numTrainVectors) {
         closest = FindNearestNeighbour(vectorSet[slot], currentBook, lastCluster + 1, TRUE, NULL);

         if (clusterMembership[slot] == -1) {      /* ie not clustered */
            avCorr  =  AverageCodebookCorrelation(vectorSet[slot],  currentBook,  lastCluster  +  1,
NULL);
            bestCorr = CMatrixCorrelationWithVect(currentBook[closest], vectorSet[slot]);
            //avCorr = CMatrixAvCorrWithinCluster(currentBook[closest]);
            stddevCorr = CMatrixStdDevCorrWithinCluster(currentBook[closest]);

            if (bestCorr > avCorr) {
            //if (fabs(bestCorr - avCorr) > (stddevCorr * clustParam)) {
               /* Make a new cluster with the current vector */
               numClusters++;
               closest = FreeCluster(currentBook, MAX_CLUSTERS);
               if (closest == -1) {
                  fprintf(stderr, "*** Exceeded compiler limit of clusters [%d].\n", MAX_CLUSTERS);
                  Abort(22);
               }
               if (closest > lastCluster)
                  lastCluster = closest;
            }

         } else {
            if (closest == clusterMembership[slot]) {      /* Not going to move */
               ignore[closest] = TRUE;
               nextClosest  =  FindNearestNeighbour(vectorSet[slot],  currentBook,  lastCluster  +  1,
TRUE, ignore);
               if (nextClosest != -1) {
                  testCorr = CMatrixNormedCorrWithVect(currentBook[nextClosest], vectorSet[slot]);
                  testDist                    =                    VectorEuclideanDist(vectorSet[slot],
CMatrixCentroid(currentBook[nextClosest]));
               } else {
                  testCorr = 9000.0;
                  testDist = 9000.0;
               }
               ignore[closest] = FALSE;

               bestCorr = CMatrixCorrelationWithVect(currentBook[closest], vectorSet[slot]);
               bestDist                    =                    VectorEuclideanDist(vectorSet[slot],
CMatrixCentroid(currentBook[closest]));

               //avCorr = AverageCodebookCorrelation(vectorSet[slot], currentBook, lastCluster + 1,
&stddevCorr);
               avCorr = CMatrixAvCorrWithinCluster(currentBook[closest]);
               stddevCorr = CMatrixStdDevCorrWithinCluster(currentBook[closest]);

               if ((currentBook[closest] -> num_vectors > 10) && ((bestCorr - avCorr) > (stddevCorr
* clustParam))) {
                  if    ((testCorr    -    CMatrixAvCorrWithinCluster(currentBook[nextClosest]))    >
(CMatrixStdDevCorrWithinCluster(currentBook[nextClosest]) * clustParam)) {
                     /* Make a new cluster with the current vector */
                     numClusters++;
                     closest = FreeCluster(currentBook, MAX_CLUSTERS);
                     if (closest == -1) {
                        fprintf(stderr,  "***  Exceeded  compiler  limit  of  clusters  [%d].\n",
MAX_CLUSTERS);
                        Abort(22);
                     }
                     if (closest > lastCluster)
                        lastCluster = closest;
```

```
                  } else {
                     closest = nextClosest;
                  }

               }
            }
         }

         oldClosest = clusterMembership[slot];

         if (oldClosest != closest) {
            /* Update the previous cluster's centroid after removing the vector */
            if (oldClosest != -1) {
               CMatrixDeleteVector(currentBook[oldClosest], vectorSet[slot]);
               if (currentBook[oldClosest] -> num_vectors == 0) {
                  currentBook[oldClosest] = CMatrixDestroy(currentBook[oldClosest]);
                  numClusters--;
               }
            }
               /* Update the current cluster's centroid after adding the vector */
            if (currentBook[closest] == NULL) {
               currentBook[closest] = CMatrixCreate(vectorSet[slot] -> dimensions);
            }
            CMatrixAddVector(currentBook[closest], VectorMakeCopy(vectorSet[slot]));
            clusterMembership[slot] = closest;
         }

         slot++;     /* Next training vector */
         if (report && (slot % (int) (numTrainVectors / 10) == 0))
            printf("|");

      }

      thisError = GetError(currentBook, lastCluster + 1);

      if (!firstRun && fabs(thisError - lastError) < INIT_STOP_PARAM) //(fabs(lastError - thisError)
/ lastError) > clustParam) /* Percentage change */
            terminate = TRUE;

      fprintf(stderr, "RMS error: %f (%5.2f%% change) [%d clusters]\n", thisError, (fabs(lastError -
thisError) / lastError) * 100.0, numClusters);
      lastError = thisError;
      firstRun = FALSE;
   }

   //ClassifyVectors(vectorSet, numTrainVectors, clusterMembership, currentBook, lastCluster, TRUE,
INIT_STOP_PARAM, NULL);

   {
      int numFinalClusters = 0;

      printf("clusters: ");
      for (slot = 0; slot <= lastCluster; slot++) {
         if (currentBook[slot]) {
            printf("%d           ",           currentBook[slot]           ->           num_vectors,
CMatrixStdDevCorrWithinCluster(currentBook[slot])/CMatrixAvCorrWithinCluster(currentBook[slot]));
            numFinalClusters++;
         }
      }
      printf("\n");
      printf("%d clusters (final), RMS err: %f\n", numFinalClusters, thisError);
   }

   /* Get rid of small clusters */

   terminate = FALSE;
   {
      double sum;

      sum = 0.0;
      for (slot = 0; slot <= lastCluster; slot++) {
         if (currentBook[slot] != NULL) {
            sum += (double) currentBook[slot] -> num_vectors;
         }
      }
      sum /= (double) numClusters;

      printf("Culling < %d\n", (int) (sum));

      while (!terminate) {
         terminate = TRUE;
         for (slot = 0; slot <= lastCluster; slot++) {
```

```
            if (currentBook[slot] != NULL) {
               if (currentBook[slot] -> num_vectors < (int) (sum))
                  ignore[slot] = TRUE;
            } else
               ignore[slot] = FALSE;
         }
         ClassifyVectors(vectorSet, numTrainVectors, clusterMembership, currentBook, lastCluster,
TRUE, INIT_STOP_PARAM, ignore);

         for (slot = 0; slot <= lastCluster; slot++) {
            if (currentBook[slot] != NULL) {
               if (currentBook[slot] -> num_vectors <  sum)
                  ;//terminate = FALSE;           /* Need another pass to cull */
            }
         }

      }
   }


   WriteCodebook(clustFile, currentBook, numTrainVectors);

   {
      int numFinalClusters = 0;

      fprintf(stderr, "clusters: ");
      for (slot = 0; slot <= lastCluster; slot++) {
         if (currentBook[slot]) {
            fprintf(stderr, "%d ", currentBook[slot] -> num_vectors);
            numFinalClusters++;
         }
      }
      fprintf(stderr, "\n");
      fprintf(stderr, "%d clusters (final), RMS err: %f\n", numFinalClusters, thisError);
   }


   /* If we are to write out the classifications, do so here */

   if (classFile != NULL) {
      printf("Writing classification.\n");
      for (slot = 0; slot < numTrainVectors; slot++) {
         fprintf(classFile, "%d\t", clusterMembership[slot]);
         VectorWrite(classFile, vectorSet[slot]);
      }
   }


   /* We need to destroy everything here! */

   CodebookDestroy(currentBook, lastCluster + 1);

   CloseFiles(vectFile, clustFile, classFile);

   return 0;
}


/* -- helper functions -- */

void Usage(FILE *output, char *appName)
{
   fprintf(output, "\n*** o2clust -- performs adaptive O2 clustering on a set of vectors\n");
   fprintf(output, "    Version %s build %s %s [%s]\n", VERSION, __TIME__, __DATE__, PLATFORM);
   fprintf(output, "Usage: %s {vector file} {cluster file} {classify file} {stop param}\n",
appName);
   fprintf(output, "Where: {vector file}   -- file to take vectors from\n");
   fprintf(output, "                          [specify '-' for stdin]\n");
   fprintf(output, "       {cluster file}  -- file to write clusters to\n");
   fprintf(output, "                          [specify '-' for stdout]\n");
   fprintf(output, "        {classify file} -- file to write classified vectors (from vector file)
to\n");
   fprintf(output, "                          [specify '-' to skip this step]\n");
   fprintf(output, "         {stop param}      -- stop clustering when RMS error changes by less
than\n");
   fprintf(output, "                                 this factor.  i.e. 0.1 => 10%%, 0.25 => 25%%,
etc.\n");
   fprintf(output, "For detailed info, type %s --help\n", appName);
   fprintf(output, "\n");
}

void AbortPrint(int code, int line, char *file)
```

```
{
   fprintf(stderr, "Aborting [%s | %d]...\n", file, line);
   exit(code);
}

void ProcessCommandLine(int argc, char *argv[], FILE **vectFile, FILE **clustFile, FILE **classFile,
double *clustParam)
{
   if ((argc > 1) && (strcasecmp(argv[HELP], "--help") == 0)) {
      FILE  *helpFile;

      if (!(helpFile = fopen(HELP_FILE, "wt"))) {
         fprintf(stderr, "*** Could not create help file [%s].\n", HELP_FILE);
         Abort(1);
      }

      Help(helpFile, argv[APP_NAME]);
      printf("Created %s\n", HELP_FILE);
      exit(0);
   }

   if (argc < NUMARGS) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Invalid number of arguments.\n");
      Abort(2);
   }

   if (argc > NUMARGS)
      fprintf(stderr, "--- Warning: extra arguments ignored.\n");

   if (strcmp(argv[VECTOR_FILE], "-") == 0)
      *vectFile = stdin;
   else if (!(*vectFile = fopen(argv[VECTOR_FILE], "rt"))) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open vector file [%s] for reading.\n", argv[VECTOR_FILE]);
      Abort(3);
   }

   if (strcmp(argv[CLUSTER_FILE], "-") == 0)
      *clustFile = stdout;
   else if (!(*clustFile = fopen(argv[CLUSTER_FILE], "wb"))) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr, "*** Could not open cluster file [%s] for writing.\n", argv[CLUSTER_FILE]);
      Abort(4);
   }

   if (strcmp(argv[CLASSIFY_FILE], "-") == 0)
      *classFile = NULL;
   else if (!(*classFile = fopen(argv[CLASSIFY_FILE], "wb"))) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr,  "***  Could  not  open  classification  file  [%s]  for  writing.\n",
argv[CLASSIFY_FILE]);
      Abort(4);
   }

   if (!isdigit(*argv[CLUST_PARAM])) {
      Usage(stderr, argv[APP_NAME]);
      fprintf(stderr,  "***  Error  converting  RMS  error  stop  parameter  from  [%s].\n",
argv[CLUST_PARAM]);
      Abort(5);
   }

   *clustParam = atof(argv[CLUST_PARAM]);
}

void CloseFiles(FILE *vectFile, FILE *clustFile, FILE *classFile)
{
   if (vectFile != stdin) fclose(vectFile);
   if (clustFile != stdout) fclose(clustFile);
   if (classFile != NULL) fclose(classFile);
}


void PrintBanner(char *argv[], FILE *vectFile, FILE *clustFile, FILE *classFile, double clustParam)
{
   fprintf(stderr, "- o2-clustering ver %s\n", VERSION);
   fprintf(stderr, " o Vector Input: << [");
   vectFile == stdin ?
      fprintf(stderr, "- stdin]\n") :
      fprintf(stderr, "%s]\n", argv[VECTOR_FILE]);
   fprintf(stderr, " o Clusters output: >> [");
   clustFile == stdout ?
```

```
      fprintf(stderr, "- stdout]\n") :
      fprintf(stderr, "%s]\n", argv[CLUSTER_FILE]);
   fprintf(stderr, " o Classification output: ");
   classFile == NULL ?
      fprintf(stderr, "skipped\n") :
      fprintf(stderr, ">> [%s]\n", argv[CLASSIFY_FILE]);
   fprintf(stderr, " o RMS error stop parameter: [%5.3f] (%3.2f%%)\n", clustParam, clustParam *
100.0);
}

void Help(FILE *helpFile, char *appName)
{
   fprintf(helpFile, "-------------------------------------\n");
   fprintf(helpFile, "   Help file for progname ver %s\n", VERSION);
   fprintf(helpFile, "-------------------------------------\n\n");

   Usage(helpFile, appName);

#include "o2clust.help.make"
}


/* -- Codebok stuff -- */

codebook CodebookNew(int slots)
{
   codebook newBook;

   if (!(newBook = (codebook) malloc(slots * sizeof(corrMatrix))))
      return NULL;

   return newBook;
}

codebook CodebookDestroy(codebook book, int slots)
{
   int     slot;

   slot = 0;
   while (slot < slots) {
      CMatrixDestroy(book[slot]);
      slot++;
   }

   free(book);
   return NULL;
}

codebook CodebookNewFromCluster(cluster *clust)
{
   codebook newBook;                /* new codebook */
   cluster  *cIndex;                /* index into cluster */
   int      slot;                   /* slot index into codebook */

   if (!(newBook = CodebookNew(ClusterSize(clust))))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      newBook[slot] = CMatrixCreate(cIndex -> vect -> dimensions);
      CMatrixAddVector(newBook[slot], cIndex -> vect);
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

codebook CodebookMakeCopy(codebook book, int slots)
{
   codebook newBook;
   int      slot;                   /* index into codebook slots */

   if (!(newBook = CodebookNew(slots)))
      return NULL;

   slot = 0;
   while (slot < slots) {
      newBook[slot] = CMatrixMakeCopy(book[slot]);
      slot++;
   }
```

```
   return newBook;
}


codebook InitialiseCodebook(int numClusters, cluster *vectorSet)
{
   codebook newBook;
   int      slot,
            numDimensions;
   cluster  *cIndex;

   if (!(newBook = CodebookNew(numClusters))) {
      fprintf(stderr, "*** Could not initialise codebok.\n");
      Abort(20);
   }

   newBook[0] = CMatrixCreate(vectorSet -> vect -> dimensions);
   CMatrixAddVector(newBook[0], vectorSet -> vect);

   for (slot = 1; slot < numClusters; slot++) {
      newBook[slot] = NULL;
   }
   return newBook;
}


cluster *ReadTrainingSet(FILE *vectorFile, int *numTrainVectors)
{
   cluster *vectorSet;

   fprintf(stderr, "Reading training vectors...\n");
   vectorSet = ClusterReadFromFile(vectorFile);
   if (ClusterIsEmpty(vectorSet)) {
      fprintf(stderr, "*** Empty training set.\n");
      Abort(10);
   }

   fprintf(stderr, "Read %d vectors.\n", ClusterSize(vectorSet));

   if  (!ClusterCheckDimensionality(vectorSet))  {             /*  make  sure  they're  all  the  same
dimensionality */
      fprintf(stderr, "*** The training vectors must all be the same dimensionality.\n");
      Abort(7);
   }
   fprintf(stderr, "Training dimension: %d\n", vectorSet -> vect -> dimensions);

   *numTrainVectors = ClusterSize(vectorSet);
   return vectorSet;
}


int  FindNearestNeighbour(vector  *vect,  codebook  currentBook,  int  numClusters,  bool  useCorr,  bool
ignore[])
{
   int      slot;          /* index into codebook */
   double   minCorr,       /* current maximum correlation (lowest absolute val) */
            corr;          /* current correlation */
   int      minSlot;       /* current slot of maximum correlation */

   minSlot = -1;
   slot = 0;
   while (slot < numClusters) {
      if (currentBook[slot] && ((ignore == NULL) || (!ignore[slot]))) {
         if (useCorr)
            corr = CMatrixNormedCorrWithVect(currentBook[slot], vect);
         else
            corr = VectorEuclideanDist(CMatrixCentroid(currentBook[slot]), vect);

         if (((minSlot == -1) || (corr < minCorr))) {
            minCorr = corr;
            minSlot = slot;
         }
      }
      slot++;
   }
   return minSlot;
}


int FreeCluster(codebook book, int numSlots)
{
```

```
   int   slot;

   for (slot = 0; slot < numSlots; slot++)
      if (book[slot] == NULL)
         return slot;

   return -1;
}

void WriteCodebook(FILE *output, codebook book, int slots)
{
   int     slot;

   for (slot = 0; slot < slots; slot++) {
      if (book[slot] != NULL) {
         CMatrixWrite(output, book[slot]);
      }
   }
}


vector **VectArrayNewFromCluster(cluster *clust)
{
   vector   **newBook;              /* new codebook */
   cluster  *cIndex;                /* index into cluster */
   int      slot;                   /* slot index into codebook */

   if (!(newBook = (vector **) malloc(ClusterSize(clust) * sizeof(vector *))))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      newBook[slot] = VectorMakeCopy(cIndex -> vect);
      cIndex = cIndex -> next;
      slot++;
   }

   return newBook;
}

double GetError(codebook book, int numClusters)
{
   double  bookError;
   int     slot;

   bookError = 0.0;
   slot = 0;
   while (slot < numClusters) {
      if (book[slot]) bookError += CMatrixRMSCorrelation(book[slot]);
      slot++;
   }

   bookError /= (double) numClusters;
   return sqrt(bookError);
}

double  AverageCodebookCorrelation(vector  *vect,  codebook  currentBook,  int  numClusters,  double
*stddev)
{
   double   corr, sum, sumSqr;
   int      slot, numReal;

   sum = 0.0;
   sumSqr = 0.0;
   numReal = 0;
   for (slot = 0; slot < numClusters; slot++) {
      if (currentBook[slot] != NULL) {
         corr = CMatrixNormedCorrWithVect(currentBook[slot], vect);
         sum += corr;
         sumSqr += corr * corr;
         numReal++;
      }
   }

   if (stddev != NULL) {
      *stddev  =  sqrt(((double)  numReal  *  sumSqr  -  (sum  *  sum))  /  (double)  (numReal  *  (numReal  -
1)));
   }
   return (sum / (double) numReal);
}
```

```
void ClassifyVectors(vector **vectorSet, int numTrainVectors, int clusterMembership[], codebook
currentBook, int lastCluster, bool useCorr, double stopParam, bool ignore[])
{
   bool     classify;         /* Do we continue classifying? */
   runAvg   *avg;             /* Running average of RMS error */
   int      slot,             /* Vector to look at */
            closest,          /* Closest cluster to  current vector */
            oldClosest,       /* Cluster vector was last in */
            tries;            /* Number of times we've tried to classify them all */
   double   closestCorr,      /* Distance to neasrest cluster */
            RMSerror,         /* current RMS error */
            lastError;

   lastError = GetError(currentBook, lastCluster + 1);
   avg = InitRunningAverage(SAMPLES_TO_AVG);
   tries = 0;
   classify = TRUE;
   while (classify) {
      slot = 0;
      while (slot < numTrainVectors) {
         closest  =  FindNearestNeighbour(vectorSet[slot],  currentBook,  lastCluster  +  1,  useCorr,
ignore);
         closestCorr = VectorEuclideanDist(currentBook[closest] -> centroid, vectorSet[slot]);
         oldClosest = clusterMembership[slot];

         if (oldClosest != closest) {
            if (oldClosest != -1) {
               CMatrixDeleteVector(currentBook[oldClosest], vectorSet[slot]);
               if (CMatrixIsEmpty(currentBook[oldClosest])) {
                  CMatrixDestroy(currentBook[oldClosest]);
                  currentBook[oldClosest] = NULL;
               }
            }

            if (!currentBook[closest])
               currentBook[closest] = CMatrixCreate(vectorSet[slot] -> dimensions);

            CMatrixAddVector(currentBook[closest], vectorSet[slot]);
            clusterMembership[slot] = closest;
         }
         slot++;
      }

      RMSerror = GetError(currentBook, lastCluster + 1);
      if (fabs(RMSerror - lastError) < stopParam)
         classify = FALSE;

      tries++;
      if (tries % 10 == 0) {
         printf(".");
         stopParam *= 2;
      }
   }
}


/* --- END of o2clust.c --- */
```

9.2.11 tlavq

Application purpose

tlavq is an extended implementation of tlearn. tlavq performs learning on a user-defined neural network, much in the same way as tlearn, except that tlavq can also perform on-line cluster analysis of specified neurons, with the intention of using this analysis to either partially or wholly classify the specified neuron's activations into another set of nodes.

The purpose of this was to implement the online clustering architecture outlined in Das and Moser [1998], but tlavq retains tlearn's inherent flexibility. The network can be configured to any architecture possible in tlearn, and clustering can be turned off entirely. With this feature disabled, the program behaves identically to tlearn.

The source code also provides an example to aid in the further extension of tlearn.

Extentions to tlearn

See section 5.0 for a discussion of the architecture proposed by Das and Moser [1998] and the modifications we made to adapt it to an Elman SRN.

The "strength" to which hidden unit activations are classified into the context layer is determined by the current learning error, modified by two parameters. These two parameters adjust eqtn (8) to provide a factor for classification strength.

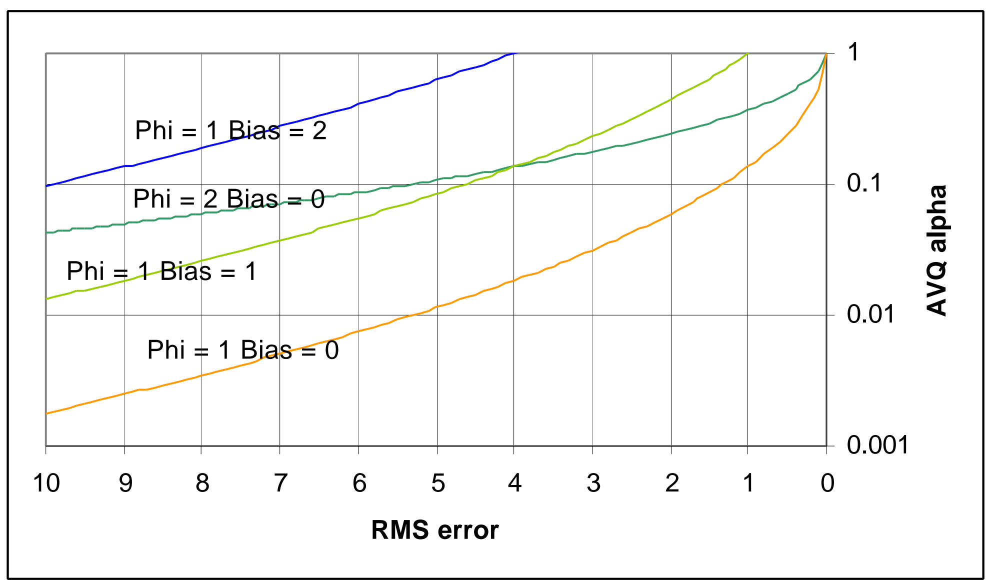


Figure 9-3 Plot of classification α vs. RMS prediction error given by eqtn (8).

$$\bar{c} = (1-\alpha)\bar{h} + \alpha\bar{i}_{\bar{h}} \qquad \textbf{Equation 10}$$

where:

- $\bar{c}$ is the vector formed by the context layer activations
- $\bar{h}$ is the vector formed by the hidden layer activations
- $\bar{i}_{\bar{h}}$ is the centroid of the cluster that $\bar{h}$ classifies to
- α is given by some function of the current prediction error (See eqtn (8)).

The format of the network configuration file is slightly different to that of tlearn. Additions have been made to the SPECIAL section to specify the nodes to be used in the clustering process:

`cluster = <node list>` defines the nodes in the network to be used as input for the cluster analysis. The activations of these nodes will be concatenated together to form a vector that defines a data point in the hidden unit space. These data points are collected over a number of epochs of learning and are subject to cluster analysis to create a corresponding codebook.

`context = <node list>` defines the nodes in the network that will form the context layer. This must be the same number of nodes as in the `cluster` node list. The classified hidden unit space vectors will be copied into these nodes in the same order as they were extracted from the hidden units.

Note that these additions do not preclude defining the recurrent network architecture in the CONNECTIONS section of the configuration file. These extra parameters are only used by the vector quantisation portion of the program.

Extra arguments have been added to the command line, to facilitate the cluster analysis:

-q:	perform AVQ on hidden unit activations to context units
-e #:	AVQ: number of sweeps in an epoch <always required for AVQ>
-p #:	AVQ: phi parameter (default = 60)
-d #:	AVQ: T parameter (default = 0.5)

The 'T' parameter is used in the adaptive Forgy's algorithm. (See section 3.1).

Modules used

StdDefs, 2DArray, Gauss, TokenLst, TokScan, Vector_Utils, Vector_Read, Cluster

Source code

avq.h

```
/* avq.h -- support header for tlearn-AVQ
 *
 * Author: Dylan Muir (dr.muir@syudent.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 18th October, 1999
 * Modified:
 * Version: 0.01
 *
 */

#ifndef  _avq_h
#define  _avq_h

#include "vector_utils.h"
#include "cluster.h"

/* -- Codebook typedef */

typedef  vector **codebook;


/* -- Codebook functions -- */

codebook CodebookNew(int slots);
codebook CodebookDestroy(codebook book, int slots);
codebook CodebookNewFromCluster(cluster *clust);
codebook CodebookMakeCopy(codebook book, int slots);
void WriteCodebook(FILE *output, codebook book, int slots);


/* -- AVQ functions -- */

int FindNearestNeighbour(vector *vect, codebook currentBook, int numSeeds);


#endif   /* _avq_h */
```

```
/* --- END of avq.h --- */
```

avq.c

```
#include <math.h>
#include "vector_utils.h"
#include "cluster.h"
#include "avq_subs.h"

#define  MAX_CLUSTERS   400
#define  AVQ_TERM_LEVEL 0.001

#if defined(ibmpc) || defined(sparc)
#define EXP(m) exp(m)
#else
#define EXP(m) \
        (exp_array[((int) ((m) * exp_mult)) + exp_add])
#endif

#define  ABS(x)   ((x) < 0 ? (0.0 - (x)) : (x))


extern   int   nn;      /* number of nodes */
extern   int   nt;      /* nn + ni + 1 size of activation arrays */
extern   int   np;      /* start of real nodes activations */
extern   int   ni;      /* number of inputs */
extern   int   nhc;     /* number of hidden/context units */

struct   cf {
   int   con;     /* connection flag */
   int   fix;     /* fixed flag */
   int   num;     /* group number */
   int   lim;     /* weight limits */
   float min;     /* weight minimum */
   float max;     /* weight maximum */
};

struct   nf {
   int   func;    /* activation function */
   int   dela;    /* delay flag */
   int   targ;    /* target flag */
};

extern   struct   cf **cinfo;
extern   struct   nf *ninfo;

extern   int   *clustnodes;      /* (nn+1) nodes for avq */
extern   int   *contxtnodes;     /* (nn+1) context nodes */
extern   int   loadflag;      /* flag to say we're loading initial state */

extern   long  epochlength;      /* number of sweeps per epoch */
extern   long  epochsforavq;     /* number of epochs each avq sweep */

extern   int   codebook_size;    /* size of codebook (number of clusters) */
extern   vector   **avq_codebook;      /* current codebook */
extern   vector   **patterns;    /* stored patterns */
extern   long  patterns_index;      /* index into patterns array */
extern   long  num_patterns;     /* number of patterns stored */

extern   double   avq_phi;    /* avq phi parameter */
extern   double   avq_t;         /* avq T parameter */

extern   float err;        /* cumulative error */
extern   float ce_err;        /* cumulative cross entropy error */
extern   float avq_err;    /* error for avq modification */

int   *clusterMembership;     /* array of what vectors belong to what clusters */


AVQ_ResetVectors()
{
   AVQ_delete_patterns();
   patterns_index = 0;
}


AVQ_classify_hu_to_context(aold, amem, anew, awt, local, atarget)
   float *aold;
   float *amem;
   float *anew;
   float **awt;
```

```
   int   *local;
   float *atarget;
{
   vector   *cluster_act;
   int   node,
      dim_index;
   double   alpha;
   float new_act;

   cluster_act  =  avq_codebook[FindNearestNeighbour(patterns[patterns_index  -  1],  avq_codebook,
codebook_size)];

   alpha = exp((0.0 - avq_phi) * avq_err);

   dim_index = 0;
   for (node = 1; node <= nn; node++) {
      if (contxtnodes[node]) {
         new_act = (float) ((1.0 - alpha) * patterns[patterns_index - 1] -> components[dim_index]);
         new_act += alpha * (double) (cluster_act -> components[dim_index]);
/*       new_act = 1.0 / (1.0 + EXP(0.0 - new_act)); */
         aold[node + ni] = anew[node + ni] = new_act;
         dim_index++;
         if (dim_index == nhc) break;
      }
   }
}

AVQ_ReClassify(void)
{
   double   lastError, thisError,
      avDist, closestDist,
      closest2Dist;
   int   slot,
      closest, oldClosest,
      closest2;
   bool  terminate;
   cluster  **clusters;

   for (slot = 1; slot < num_patterns; slot++)
      clusterMembership[slot] = 0;
   clusterMembership[0] = 1;

   avq_codebook = CodebookDestroy(avq_codebook, codebook_size);
   avq_codebook = InitialiseCodebook(MAX_CLUSTERS, patterns, num_patterns);
   clusters = MakeClusters(MAX_CLUSTERS);
   clusters[0] = ClusterFromCodebook(patterns, num_patterns);
   clusters[0] = ClusterRemoveVector(clusters[0], patterns[0]);
   codebook_size = 2;
   lastError = 999.0;

#ifdef   DEBUG
   fprintf(stderr, "AVQ:(%4.2f)[", avq_t);
   fflush(stdout);
#endif

   terminate = FALSE;
   while (!terminate) {
      slot = 0;
      while (slot < num_patterns) {
         avDist = CodebookAverageDistance(patterns[slot], avq_codebook, codebook_size);
         closest = FindNearestNeighbour(patterns[slot], avq_codebook, codebook_size);
         closestDist = VectorEuclideanDist(patterns[slot], avq_codebook[closest]);
/*       closest2 = FindNearestOtherCluster(avq_codebook, codebook_size, closest);
         closest2Dist = VectorEuclideanDist(avq_codebook[closest], avq_codebook[closest2]); */

         if (closestDist > (avDist * avq_t)) {
/*       if (ABS(closestDist - avDist) < (avDist * avq_t)) { */
/*       if (closestDist > (avq_t * closest2Dist)) { */
            closest = FreeCluster(avq_codebook, MAX_CLUSTERS);

            if (closest == -1) {
               perror("ERROR: Exceeded compiler limit on clusters");
               exit(1);
            }
            if (closest >= codebook_size)
               codebook_size = closest + 1;
         }

         oldClosest = clusterMembership[slot];
         clusters[oldClosest] = ClusterRemoveVector(clusters[oldClosest], patterns[slot]);
         if (avq_codebook[oldClosest] != NULL)
            VectorDeallocate(avq_codebook[oldClosest]);
```

```
         avq_codebook[oldClosest] = ClusterCentroid(clusters[oldClosest]);

         clusters[closest] = ClusterAddVector(clusters[closest], VectorMakeCopy(patterns[slot]));
         clusterMembership[slot] = closest;
         if (avq_codebook[closest] != NULL)
            VectorDeallocate(avq_codebook[closest]);
         avq_codebook[closest] = ClusterCentroid(clusters[closest]);

         slot++;
      }

      thisError = GetError(clusters, avq_codebook, codebook_size);
      if  ((codebook_size  >=  20)  &&  (lastError  >=  thisError)  &&  (lastError  -  thisError  <
AVQ_TERM_LEVEL))
         terminate = TRUE;
      lastError = thisError;
      if (codebook_size < 20) {
         avq_t -= 0.01;
#ifdef   DEBUG
         fprintf(stderr, "-");
#endif
      } else if (codebook_size > 80) {
         avq_t += 0.01;
#ifdef   DEBUG
         fprintf(stderr, "+");
#endif
      } else {
#ifdef   DEBUG
         fprintf(stderr, ".");
#endif
      }
#ifdef   DEBUG
      fflush(stdout);
#endif
   }

#ifdef   DEBUG
   fprintf(stderr, "] RMS error: %f (%d clusters)\n", thisError, codebook_size);
   fflush(stdout);
#endif

   clusters = DestroyClusters(clusters, MAX_CLUSTERS);
}


AVQ_check_hidden_context(void)
{
   int   node;
   int   count_clust, count_contxt;

   count_clust = 0;
   for (node = 1; node <= nn; node++)
      if (clustnodes[node]) count_clust++;

   count_contxt = 0;
   for (node = 1; node <= nn; node++)
      if (contxtnodes[node]) count_contxt++;

   if (count_contxt != count_clust) {
      perror("ERROR: number of context nodes must equal the number of cluster nodes");
      exit(1);
   }

   nhc = count_contxt;
}

AVQ_initialise(void)
{
   int   dIndex;

   if (nhc == 0) {
      perror("ERROR: num context/cluster units is zero!");
      exit(1);
   }

   if (!(patterns = (codebook) malloc(num_patterns * sizeof(vector *)))) {
      perror("ERRPR: could not allocate patterns array\n");
      exit(1);
   }
   AVQ_wipe_patterns();
   patterns_index = 0;
```

```
   if (!loadflag) {
      if (!(avq_codebook = CodebookNew(MAX_CLUSTERS))) {
         perror("ERROR: could not allocate initial codebook");
         exit(1);
      }

      avq_codebook[0] = VectorAllocate(nhc); /* initial cluster is zero */
      for (dIndex = 0; dIndex < nhc; dIndex++)
         avq_codebook[0] -> components[dIndex] = 0.5;

      codebook_size = 1;
   }

   if (!(clusterMembership = (int *) malloc(num_patterns * sizeof(int)))) {
      perror("ERROR: could not allocate cluster membership array");
      exit(1);
   }
}

AVQ_add_pattern(aold)
   float *aold;
{
   vector   *hu_activation;
   int   node,
      dimension_index;

   if (!(hu_activation = VectorAllocate(nhc))) {
      perror("ERROR: unable to allocate pattern");
      exit(1);
   }

   dimension_index = 0;
   for (node = 1; node <= nn; node++) {
      if (clustnodes[node]) {
         hu_activation -> components[dimension_index] = aold[node + ni];
         dimension_index++;
         if (dimension_index == nhc) break;
      }
   }

   patterns[patterns_index] = hu_activation;
   patterns_index++;
}


AVQ_wipe_patterns(void)
{
   int   pattern_index;

   for (pattern_index = 0; pattern_index < num_patterns; pattern_index++)
      patterns[pattern_index] = NULL;
}

AVQ_delete_patterns(void)
{
   int   pattern_index;

   for (pattern_index = 0; pattern_index < num_patterns; pattern_index++)
      patterns[pattern_index] = VectorDeallocate(patterns[pattern_index]);
}


/* --- END of avq.c --- */
```

avq_subs.h

```
/* avq_subs.h -- SUbroutines for AVQ
 */

#ifndef  _avq_subs_h
#define  _avq_subs_h

#include <stdio.h>
#include "stddefs.h"
#include "vector_utils.h"
#include "cluster.h"

typedef  vector   **codebook;

bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize);
bool CheckClusterDimensions(cluster *clust);
```

```
codebook ReadTrainingSet(FILE *vectorFile, int *numVectors);
cluster **MakeClusters(int numClusters);
cluster **DestroyClusters(cluster **clusters, int numClusters);
int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters);
int FindNearestOtherCluster(codebook currentBook, int numClusters, int target);
double GetError(cluster **clusters, codebook book, int numClusters);
double ClusterRMSDistance(cluster *clust, vector *vect);
codebook InitialiseCodebook(int numClusters, codebook vectorSet, int numSlots);
double CodebookAverageDistance(vector *vect, codebook book, int numClusters);
vector *CodebookCentroid(codebook book, int numSlots);
int FreeCluster(codebook book, int numSlots);
cluster *ClusterFromCodebook(codebook book, int numSlots);


/* -- Codebook functions -- */
codebook CodebookNew(int slots);
codebook CodebookDestroy(codebook book, int slots);
codebook CodebookNewFromCluster(cluster *clust, int *slots);
codebook CodebookMakeCopy(codebook book, int slots);
void WriteCodebook(FILE *output, codebook book, int slots);


#endif /* _avq_subs_h */
```

avq_subs.c

```
/* avq_subs.c -- Subroutines for AVQ
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "avq_subs.h"
#include "stddefs.h"
#include "vector_utils.h"
#include "vector_read.h"
#include "cluster.h"

#define Abort(x)        AbortPrint((x), __LINE__, __FILE__);
#define SQR(x)          ((x) * (x))

void AbortPrint(int code, int line, char *file)
{
   fprintf(stderr, "Aborting... [%d | %s]\n", line, file);
   exit(code);
}


bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize)
{
   int   index;

   if ((*array = malloc(xSize * sizeof(void *))) == NULL)
      return FALSE;

   for (index = 0; index < xSize; index++) {
      if (((*array)[index] = malloc(ySize * elementSize)) == NULL)
         return FALSE;

      xSize--;
   }
}

codebook CodebookNew(int slots)
{
   codebook newBook;

   if (!(newBook = (vector **) malloc(slots * sizeof(vector *))))
      return NULL;

   return newBook;
}

bool CheckClusterDimensions(cluster *clust)
{
   int      dimensions;             /* dimension to check against */
   cluster  *cIndex;                /* index into cluster */

   if (clust == NULL) return TRUE; /* empty cluster, so we pass */

   dimensions = clust -> vect -> dimensions;
```

```
   cIndex = clust -> next;         /* the first one passes by definition */

   while (cIndex != NULL) {
      if (cIndex -> vect -> dimensions != dimensions)
         return FALSE;

      cIndex = cIndex -> next;
   }

   return TRUE;
}

codebook CodebookNewFromCluster(cluster *clust, int *slots)
{
   codebook newBook;                /* new codebook */
   cluster  *cIndex;                /* index into cluster */
   int      slot;                   /* slot index into codebook */

   if (!(newBook = CodebookNew(ClusterSize(clust))))
      return NULL;

   slot = 0;
   cIndex = clust;
   while (cIndex != NULL) {
      newBook[slot] = VectorMakeCopy(cIndex -> vect);
      cIndex = cIndex -> next;
      slot++;
   }

   *slots = slot;

   return newBook;
}

codebook CodebookDestroy(codebook book, int slots)
{
   int   slot;

   slot = 0;
   while (slot < slots) {
      VectorDeallocate(book[slot]);
      slot++;
   }

   free(book);
   return NULL;
}


codebook CodebookMakeCopy(codebook book, int slots)
{
   codebook newBook;
   int      slot;                   /* index into codebook slots */

   if (!(newBook = CodebookNew(slots)))
      return NULL;

   slot = 0;
   while (slot < slots) {
      newBook[slot] = VectorMakeCopy(book[slot]);
      slot++;
   }

   return newBook;
}

cluster **MakeClusters(int numSeeds)
{
   cluster **newClusterArray;
   int     slot;                   /* index into cluster array */

   if (!(newClusterArray = (cluster **) malloc(numSeeds * sizeof(cluster *)))) {
      fprintf(stderr, "*** Unable to allocate cluster space.\n");
      Abort(14);
   }

   slot = 0;
   while (slot < numSeeds) {
      newClusterArray[slot] = ClusterNew();
      slot++;
   }
```

```
   return newClusterArray;
}

cluster **DestroyClusters(cluster **clusters, int numSeeds)
{
   int   slot;

   slot = 0;
   while (slot < numSeeds) {
      clusters[slot] = ClusterDestroy(clusters[slot]);
      slot++;
   }

   free(clusters);

   return NULL;
}

int FindNearestOtherCluster(codebook currentBook, int numClusters, int target)
{
   int   slot, minSlot;
   double   minDist,
      dist;
   vector   *targetVect;

   minSlot = 0;
   targetVect = currentBook[target];
   slot = 0;

   if (target != 0)
      minSlot = 0;
   else if (numClusters == 1)
      return 0;
   else
      minSlot = 1;

   minDist = VectorEuclideanDist(currentBook[target], currentBook[minSlot]);

   while (slot < numClusters) {
      if (slot != target) {
         if (currentBook[slot] != NULL) {
            dist = VectorEuclideanDist(targetVect, currentBook[slot]);
            if (dist < minDist) {
               minDist = dist;
               minSlot = slot;
            }
         }
      }
      slot++;
   }

   return minSlot;
}


int FindNearestNeighbour(vector *vect, codebook currentBook, int numClusters)
{
   int      slot;           /* index into codebook */
   double   minDist,        /* current minimum distance */
            dist;           /* current distance */
   int      minSlot;        /* current slot of minimum distance*/

   minSlot = 0;
   minDist = VectorEuclideanDist(vect, currentBook[0]);
   slot = 1;
   while (slot < numClusters) {
      if (currentBook[slot] != NULL) {
         dist = VectorEuclideanDist(vect, currentBook[slot]);

         if (dist < minDist) {
            minDist = dist;
            minSlot = slot;
         }
      }
      slot++;
   }

   return minSlot;
}

double GetError(cluster **clusters, codebook book, int numClusters)
{
```

```
   double   bookError;
   int      slot;

   bookError = 0.0;
   slot = 0;
   while (slot < numClusters) {
      bookError += ClusterRMSDistance(clusters[slot], book[slot]);
      slot++;
   }

   bookError /= (double) numClusters;
   return sqrt(bookError);
}

double ClusterRMSDistance(cluster *clust, vector *vect)
{
   double   errSum;
   cluster  *cIndex;

   if (ClusterIsEmpty(clust))
      return 0.0;

   errSum = 0.0;
   cIndex = clust;
   while (cIndex != NULL) {
      errSum += SQR(VectorEuclideanDist(vect, cIndex -> vect));
      cIndex = cIndex -> next;
   }

   errSum /= (double) ClusterSize(clust);
   return sqrt(errSum);
}

double CodebookAverageDistance(vector *vect, codebook book, int numClusters)
{
   double   distance;
   int      slot;

   distance = 0.0;
   slot = 0;
   while (slot < numClusters) {
      if (book[slot] == NULL) {
         numClusters--;
         slot++;
         continue;
      }
      distance += VectorEuclideanDist(vect, book[slot]);
      slot++;
   }

   return distance / (double) numClusters;
}

void WriteCodebook(FILE *output, codebook book, int slots)
{
   int   slot;

   for (slot = 0; slot < slots; slot++) {
      if (book[slot] != NULL) {
         VectorWrite(output, book[slot]);
         fprintf(output, "\n");
      }
   }
}


codebook InitialiseCodebook(int numClusters, codebook vectorSet, int numSlots)
{
   codebook newBook;
   int      slot;

   if (!(newBook = CodebookNew(numClusters))) {
      fprintf(stderr, "*** Could not initialise codebok.\n");
      Abort(20);
   }

   if (!(newBook[0] = CodebookCentroid(vectorSet, numSlots))) {
      fprintf(stderr, "*** Could not create initial centroid vector.\n");
      Abort(21);
   }

   if (!(newBook[1] = VectorMakeCopy(vectorSet[0]))) {
```

```
      fprintf(stderr, "*** Could not allocate initial pattern vector.\n");
      Abort(23);
   }

   for (slot = 2; slot < numClusters; slot++)
      newBook[slot] = NULL;

   return newBook;
}

vector *CodebookCentroid(codebook book, int numSlots)
{
   vector   *vectSum;
   int      slot, entries;

   if (numSlots == 0)
      return NULL;

   if (!(vectSum = VectorZero(book[0] -> dimensions)))
      return NULL;

   entries = 0;
   slot = 0;
   while (slot < numSlots) {
      if (book[slot] != NULL) {
         VectorSum(vectSum, book[slot]);
         entries++;
      }
      slot++;
   }

   VectorDivideScalar(vectSum, (double) entries);
   return vectSum;
}

int FreeCluster(codebook book, int numSlots)
{
   int   slot;

   for (slot = 0; slot < numSlots; slot++)
      if (book[slot] == NULL)
         return slot;

   return -1;
}

cluster *ClusterFromCodebook(codebook book, int numSlots)
{
   cluster  *newClust;
   int   slot;

   newClust = NULL;
   slot = 0;

   while (slot < numSlots) {
      if (book[slot] != NULL)
         newClust = ClusterAddVector(newClust, VectorMakeCopy(book[slot]));
      slot++;
   }

   return newClust;
}

/* --- END of avq_subs.c --- */
```

tlavq.c

```
/* tlavq.c - simulator for arbitrary networks with time-ordered input */
/*           Includes vector quantisation -- Dylan Muir October 1999  */

/*----------------------------------------------------------------------

This program simulates learning in a neural network using either
the classical back-propagation learning algorithm or a slightly
modified form derived in Williams and Zipser, "A Learning Algo-
rithm for Continually Running Fully Recurrent Networks."  The
input is a sequence of vectors of (ascii) floating point numbers
contained in a ".data" file.  The target outputs are a set of
time-stamped vectors of (ascii) floating point numbers (including
optional "don't care" values) in a ".teach" file.  The network
configuration is defined in a ".cf" file documented in tlearn.man.
```

```
---------------------------------------------------------------------*/

#include <math.h>
#include <stdio.h>
#include <signal.h>
#ifdef ibmpc
#include "strings.h"
#include <fcntl.h>
#else
#include <string.h>
#include <sys/file.h>
#endif
#include <sys/types.h>
#include <sys/stat.h>

#include "vector_utils.h"
#include "cluster.h"
#include "avq.h"

#ifdef ibmpc
#define random(x) rand(x)
#define srandom(x) srand(x)
#endif

int   nn;      /* number of nodes */
int   ni;      /* number of inputs */
int   no;      /* number of outputs */
int   nt;      /* nn + ni + 1 */
int   np;      /* ni + 1 */
int   nhc;     /* number of hidden/context units */

struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight-limits flag */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};

struct   cf **cinfo; /* (nn x nt) connection info */
struct   nf *ninfo;     /* (nn) node activation function info */

int   *outputs;   /* (no) indices of output nodes */
int   *selects;   /* (nn+1) nodes selected for probe printout */
int   *linput; /* (ni) localist input array */
int   *clustnodes;   /* (nn+1) nodes to use avq on */
int   *contxtnodes;  /* (nn+1) nodes defined as context units */

long  epochlength = 0;  /* number of sweeps in one epoch */
long  epochsforavq = 2; /* number of epochs to gather for avq */
long  currentepoch = 0; /* current epoch we're in */
int   codebook_size;    /* size of codebook (number of clusters */
vector   **avq_codebook;      /* current AVQ codebook */
vector   **patterns;    /* HU patterns */
long  patterns_index = 0;  /* index into patterns array */
long  num_patterns;     /* max number of patterns */

double   avq_phi = 60;  /* 'phi' parameter for avq */
double   avq_t = 0.5;   /* 'T' parameter for avq */

float *znew;      /* (nt) inputs and activations at time t+1 */
float *zold;      /* (nt) inputs and activations at time t */
float *zmem;      /* (nt) inputs and activations at time t */
float **wt;    /* (nn x nt) weight TO node i FROM node j*/
float **dwt;      /* (nn x nt) delta weight at time t */
float **winc;     /* (nn x nt) accumulated weight increment*/
float *target; /* (no) output target values */
float *error;     /* (nn) error = (output - target) values */
float ***pnew; /* (nn x nt x nn) p-variable at time t+1 */
float ***pold; /* (nn x nt x nn) p-variable at time t */

float rate = .1;  /* learning rate */
float momentum = 0.; /* momentum */
float weight_limit = 1.;   /* bound for random weight init */
```

```
float criterion = 0.;   /* exit program when rms error is less than this */
float init_bias = 0.;   /* possible offset for initial output biases */

float err = 0.;   /* cumulative ss error */
float ce_err = 0.;   /* cumulate cross_entropy error */
float avq_err; /* error for avq modification */

long  sweep = 0;  /* current sweep */
long  err_sweeps = 0;   /* number of sweeps for cumulative error */
long  tsweeps = 0;   /* total sweeps to date */
long  report = 0; /* output rms error every "report" sweeps */

int   ngroups = 0;   /* number of groups */

int   backprop = 1;  /* flag for standard back propagation (the default) */
int   teacher = 0;   /* flag for feeding back targets */
int   localist = 0;  /* flag for speed-up with localist inputs */
int   randomly = 0;  /* flag for presenting inputs in random order */
int   limits = 0; /* flag for limited weights */
int   ce = 0;     /* flag for cross_entropy */
int   avq_flag = 0;  /* flag to perform avq */
int   loadflag = 0;  /* flag for loading initial weights from file */

char  root[128];  /* root filename for .cf, .data, .teach, etc.*/
char  loadfile[128]; /* filename for weightfile to be read in */

FILE  *cfp;    /* file pointer for .cf file */

void intr();

extern int load_wts();
extern int save_wts();
extern int act_nds();


main(argc,argv)
   int   argc;
   char  **argv;
{

   FILE  *fopen();
   extern   char *optarg;
   extern   float rans();
   extern   long time();
/* extern   int intr(); */


   long  nsweeps = 0;   /* number of sweeps to run for */
   long  ttime = 0;  /* number of sweeps since time = 0 */
   long  utime = 0;  /* number of sweeps since last update_weights */
   long  tmax = 0;   /* maximum number of sweeps (given in .data) */
   long  umax = 0;   /* update weights every umax sweeps */
   long  rtime = 0;  /* number of sweeps since last report */
   long  check = 0;  /* output weights every "check" sweeps */
   long  ctime = 0;  /* number of sweeps since last check */

   int   c;
   int   i;
   int   j;
   int   k;
   int   nticks = 1; /* number of internal clock ticks per input */
   int   ticks = 0;  /* counter for ticks */
   int   learning = 1;  /* flag for learning */
   int   reset = 0;  /* flag for resetting net */
   int   verify = 0; /* flag for printing output values */
   int   probe = 0;  /* flag for printing selected node values */
   int   command = 1;   /* flag for writing to .cmd file */
   int   iflag = 0;  /* flag for -I */
   int   tflag = 0;  /* flag for -T */
   int   rflag = 0;  /* flag for -x */
   int   seed = 0;   /* seed for random() */
   float avq_err_bias = 0.0;  /* offset for alpha calculation */

   float *w;
   float *wi;
   float *dw;
   float *pn;
   float *po;

   struct   cf *ci;

   char  cmdfile[128];  /* filename for logging runs of program */
```

```
   char  cfile[128]; /* filename for .cf file */

   FILE  *cmdfp;

   signal(SIGINT, intr);
#ifndef ibmpc
   signal(SIGHUP, intr);
   signal(SIGQUIT, intr);
   signal(SIGKILL, intr);
#endif

#ifndef ibmpc
   exp_init();
#endif

#ifdef   DEBUG
   fprintf(stderr,"TlearnAVQ build [%s %s] DEBUG\n", __DATE__, __TIME__);
#endif

   root[0] = 0;
   nhc = 0;

   while ((c = getopt(argc, argv, "o:f:hiql:m:n:r:s:e:d:p:tC:E:ILM:PRS:TU:VXB:H:")) != EOF) {
      switch (c) {
         case 'o':
            avq_err_bias = atof(optarg);
#ifdef   DEBUG
            fprintf(stderr, "AVQ RMS error bias: %f\n", avq_err_bias);
#endif
            break;
         case 'q':
            avq_flag = 1;
#ifdef   DEBUG
            fprintf(stderr, "Performing AVQ on hidden units\n");
#endif
            break;
         case 'e':
            epochlength = (long) atol(optarg);
#ifdef   DEBUG
            fprintf(stderr, "Sweeps per epoch: %ld\n", epochlength);
#endif
            break;
         case 'p':
            avq_phi = (double) atof(optarg);
#ifdef   DEBUG
            fprintf(stderr, "AVQ Phi parameter: %f\n", avq_phi);
#endif
            break;
         case 'd':
            avq_t = (double) atof(optarg);
#ifdef   DEBUG
            fprintf(stderr, "AVQ T parameter: %f\n", avq_t);
#endif
            break;
         case 'C':
            check = (long) atol(optarg);
            ctime = check;
#ifdef   DEBUG
            fprintf(stderr, "Weights checkpoint: %ld\n", check);
#endif
            break;
         case 'f':
            strcpy(root,optarg);
#ifdef   DEBUG
            fprintf(stderr, "Input file base: [%s]\n", optarg);
#endif
            break;
         case 'i':
            command = 0;
            break;
         case 'l':
            loadflag = 1;
            strcpy(loadfile,optarg);
#ifdef   DEBUG
            fprintf(stderr, "Load state file base: [%s].wts(, .means)\n", optarg);
#endif
            break;
         case 'm':
            momentum = (float) atof(optarg);
#ifdef   DEBUG
            fprintf(stderr, "Momentum: %f\n", momentum);
#endif
```

```
            break;
        case 'n':
            nticks = (int) atoi(optarg);
#ifdef  DEBUG
            fprintf(stderr, "Clock ticks per input: %d\n", nticks);
#endif
            break;
        case 'P':
            probe = 1;
            learning = 0;
#ifdef  DEBUG
            fprintf(stderr, "Probe mode on\n");
#endif
            break;
        case 'r':
            rate = (float) atof(optarg);
#ifdef  DEBUG
            fprintf(stderr, "Learning rate: %f\n", rate);
#endif
            break;
        case 's':
            nsweeps = (long) atol(optarg);
#ifdef  DEBUG
            fprintf(stderr, "Sweeps to execute: %ld\n", nsweeps);
#endif
            break;
        case 't':
            teacher = 1;
#ifdef  DEBUG
            fprintf(stderr, "Feedback teacher values mode on\n");
#endif
            break;
        case 'L':
            backprop = 0;
#ifdef  DEBUG
            fprintf(stderr, "RTRL mode on\n");
#endif
            break;
        case 'V':
            verify = 1;
            learning = 0;
#ifdef  DEBUG
            fprintf(stderr, "Verify mode on\n");
#endif
            break;
        case 'X':
            rflag = 1;
#ifdef  DEBUG
            fprintf(stderr, "Using .reset file\n");
#endif
            break;
        case 'E':
            report = (long) atol(optarg);
#ifdef  DEBUG
            fprintf(stderr, "RMS checkpoint: %ld\n", report);
#endif
            break;
        case 'I':
            iflag = 1;
#ifdef  DEBUG
            fprintf(stderr, "Ignoring input values during extra clock ticks\n");
#endif
            break;
        case 'M':
            criterion = (float) atof(optarg);
#ifdef  DEBUG
            fprintf(stderr, "RMS target value for termination: %f\n", criterion);
#endif
            break;
        case 'R':
            randomly = 1;
#ifdef  DEBUG
            fprintf(stderr, "Randomly presenting input patterns\n");
#endif
            break;
        case 'S':
            seed = atoi(optarg);
#ifdef  DEBUG
            fprintf(stderr, "RNG seed: %d\n", seed);
#endif
            break;
        case 'T':
```

```
            tflag = 1;
#ifdef   DEBUG
            fprintf(stderr, "Ignoring target values during extra clock ticks\n");
#endif
            break;
         case 'U':
            umax = atol(optarg);
            break;
         case 'B':
            init_bias = atof(optarg);
#ifdef   DEBUG
            fprintf(stderr, "Initial bias offset: %f\n", init_bias);
#endif
            break;
         /*
          * if == 1, use cross-entropy as error;
          * if == 2, also collect cross-entropy stats.
          */
         case 'H':
            ce = atoi(optarg);
            break;
         case '?':
         case 'h':
         default:
            usage();
            exit(2);
         break;
      }
   }

   if (nsweeps == 0){
      perror("ERROR: No -s specified");
      exit(1);
   }

   if (avq_flag && (epochlength == 0)) {
      perror("ERROR: No -e specified for AVQ mode");
      exit(1);
   }

   /* open files */

   if (root[0] == 0){
      perror("ERROR: No fileroot specified");
      exit(1);
   }

   if (command) {
      sprintf(cmdfile, "%s.cmd", root);
      cmdfp = fopen(cmdfile, "a");
      if (cmdfp == NULL) {
         perror("ERROR: Can't open .cmd file");
         exit(1);
      }
      for (i = 1; i < argc; i++)
         fprintf(cmdfp,"%s ",argv[i]);
      fprintf(cmdfp,"\n");
      fflush(cmdfp);
   }

   sprintf(cfile, "%s.cf", root);
   cfp = fopen(cfile, "r");
   if (cfp == NULL) {
      perror("ERROR: Can't open .cf file");
      exit(1);
   }

   get_nodes();
   make_arrays();
   get_outputs();
   get_connections();
   get_special();

   if (nhc == 0) {
      perror("ERROR: Hidden and context nodes must be specified");
      exit(1);
   }

   if (avq_flag) AVQ_check_hidden_context();

   if (!seed)
      time(&seed);
```

```
   srandom(seed);

   if (loadflag)
      load_wts();
   else {
      for (i = 0; i < nn; i++){
         w = *(wt + i);
         dw = *(dwt+ i);
         wi = *(winc+ i);
         ci = *(cinfo+ i);
         for (j = 0; j < nt; j++, ci++, w++, wi++, dw++){
            if (ci->con)
               *w = rans(weight_limit);
            else
               *w = 0.;
            *wi = 0.;
            *dw = 0.;
         }
      }
      /*
       * If init_bias, then we want to set initial biases
       * to (*only*) output units to a random negative number.
       * We index into the **wt to find the section of receiver
       * weights for each output node.  The first weight in each
       * section is for unit 0 (bias), so no further indexing needed.
       */
      for (i = 0; i < no; i++){
         w = *(wt + outputs[i] - 1);
         ci = *(cinfo + outputs[i] - 1);
         if (ci->con)
            *w = init_bias + rans(.1);
         else
            *w = 0.;
      }
   }
   zold[0] = znew[0] = 1.;
   for (i = 1; i < nt; i++)
      zold[i] = znew[i] = 0.;
   if (backprop == 0){
      make_parrays();
      for (i = 0; i < nn; i++){
         for (j = 0; j < nt; j++){
            po = *(*(pold + i) + j);
            pn = *(*(pnew + i) + j);
            for (k = 0; k < nn; k++, po++, pn++){
               *po = 0.;
               *pn = 0.;
            }
         }
      }
   }


   nsweeps += tsweeps;
   sweep = tsweeps;
   err_sweeps = 0;
   currentepoch = 0;
   while (sweep < nsweeps) {

/* for (sweep = tsweeps; sweep < nsweeps; sweep++){*/
      for (ticks = 0; ticks < nticks; ticks++){

         update_reset(ttime,ticks,rflag,&tmax,&reset);

         if (reset){
            if (backprop == 0)
               reset_network(zold,znew,pold,pnew);
            else
               reset_bp_net(zold,znew);
         }

         update_inputs(zold,ticks,iflag,&tmax,&linput);

         if (avq_flag && (sweep == tsweeps)) {     /* need to initialise AVQ */
            num_patterns = tmax * epochsforavq;
            AVQ_initialise();
         }

         if (learning || teacher || (report != 0))
            update_targets(target,ttime,ticks,tflag,&tmax);

         act_nds(zold,zmem,znew,wt,linput,target);
```

```
         if (learning) {   /* modified to always calculate error */
            comp_errors(zold,target,error,&err,&ce_err);
            avq_err = sqrt((err / (double) err_sweeps + 1.0) / (double) no) - avq_err_bias;
         }  /* Note: by here, we've really done another sweep */
            /* avq_err = (sqrt(mean_err) / sqrt(num outputs)) - bias */

         if (avq_flag) {
            if (patterns_index < num_patterns) AVQ_add_pattern(zold);
            AVQ_classify_hu_to_context(zold, zmem, znew, wt, linput, target);
         }

         if (learning && (backprop == 0))
            comp_deltas(pold,pnew,wt,dwt,zold,znew,error);
         if (learning && (backprop == 1))
            comp_backprop(wt,dwt,zold,zmem,target,error,linput);

         if (probe)
            print_nodes(zold);
      }

      if (avq_flag && (sweep % epochlength == 0) && (sweep > tsweeps)) {
         currentepoch++;
         if (currentepoch >= epochsforavq) {
            AVQ_ReClassify();
            currentepoch = 0;
            AVQ_ResetVectors();
         }
      }

      if (verify)
         print_output(zold);

      if (report && (++rtime >= report)){
         rtime = 0;
#ifdef   DEBUG
         if (avq_flag)
            fprintf(stderr, "err:%f alpha:%f\n", avq_err, exp(-avq_phi * avq_err));
#endif
         if (ce == 2)
            print_error(&ce_err);
         else
            print_error(&err);
      }

      if (check && (++ctime >= check)){
         ctime = 0;
         save_wts();
      }

      if (++ttime >= tmax)
         ttime = 0;

      if (++utime >= umax){
         utime = 0;
         update_weights(wt,dwt,winc);
      }

      sweep++;
      err_sweeps++;
   }
   if (learning)
      save_wts();
   exit(0);

}

usage() {
  fprintf(stderr, "\n");
  fprintf(stderr, "-f fileroot:\tspecify fileroot <always required>\n");
  fprintf(stderr, "-l weightfile:\tload in weightfile\n");
  fprintf(stderr, "\n");
  fprintf(stderr, "-s #:\trun for # sweeps <always required>\n");
  fprintf(stderr, "-q:\tperform AVQ on hidden unit activations to context units\n");
  fprintf(stderr, "-e #:\tAVQ: number of sweeps in an epoch <always required>\n");
  fprintf(stderr, "-p #:\tAVQ: phi parameter (default = 60)\n");
  fprintf(stderr, "-d #:\tAVQ: T parameter (default = 0.5)\n");
  fprintf(stderr, "-r #:\tset learning rate to # (between 0. and 1.) [0.1]\n");
  fprintf(stderr, "-m #:\tset momentum to # (between 0. and 1.) [0.0]\n");
  fprintf(stderr, "-n #:\t# of clock ticks per input vector [1]\n");
  fprintf(stderr, "-t:\tfeedback teacher values in place of outputs\n");
  fprintf(stderr, "\n");
```

```
  fprintf(stderr, "-S #:\tseed for random number generator [random]\n");
  fprintf(stderr, "-U #:\tupdate weights every # sweeps [1]\n");
  fprintf(stderr, "-E #:\trecord rms error in .err file every # sweeps [0]\n");
  fprintf(stderr, "-C #:\tcheckpoint weights file every # sweeps [0]\n");
  fprintf(stderr, "-M #:\texit program when rms error is less than # [0.0]\n");
  fprintf(stderr, "-X:\tuse auxiliary .reset file\n");
  fprintf(stderr, "-P:\tprobe selected nodes on each sweep (no learning)\n");
  fprintf(stderr, "-V:\tverify outputs on each sweep (no learning)\n");
  fprintf(stderr, "-R:\tpresent input patterns in random order\n");
  fprintf(stderr, "-I:\tignore input values during extra clock ticks\n");
  fprintf(stderr, "-T:\tignore target values during extra clock ticks\n");
  fprintf(stderr, "-L:\tuse RTRL temporally recurrent learning\n");
  fprintf(stderr, "-B #:\toffset for offset biasi initialization\n");
  fprintf(stderr, "\n");
}

void
intr(sig)
   int   sig;
{
   save_wts();
   exit(sig);
}
```

weights.c

```
#include <stdio.h>
#include "vector_utils.h"
#include "avq_subs.h"
#include "cluster.h"

extern   int   nn;   /* number of nodes */
extern   int   ni;   /* number of inputs */
extern   int   nt;   /* nn + ni + 1 */

extern   long  tsweeps; /* total sweeps */
extern   long  sweep;      /* current sweep */

extern   float **wt;    /* (nn x nt): weights */

extern   char  loadfile[]; /* state file base to start with */
extern   char  root[];     /* root file name */

extern   int      avq_flag;   /* flag to perform avq */
extern   vector** avq_codebook;  /* current AVQ codebook */
extern   int      codebook_size;

save_wts()
{

   FILE  *fp;
   FILE  *fopen();

   register int   i;
   register int   j;

   float *w;
   float **wp;

   char  weights_file[128],
      means_file[128];

#ifdef ibmpc
   /*
    * if running under DOS, probably can't have filenames
    * with more than 8 chars total, or multiple "."s, so
    * "fileroot.nnnnn.wts" becomes "fileroot_nnnnn.wts"
    */
   sprintf(weights_file, "%s_%ld.wts", root, sweep);
   sprintf(means_file, "%s_%ld.mns", root, sweep);
#else
   sprintf(weights_file, "%s.%ld.wts", root, sweep);
   sprintf(means_file, "%s.%ld.means", root, sweep);
#endif
   if ((fp=fopen(weights_file, "w+")) == NULL) {
      perror("Can't open .wts file\n");
      exit(1);
   }
   fprintf(fp, "NETWORK CONFIGURED BY TLEARN\n");
   fprintf(fp, "# weights after %ld sweeps\n", sweep);
   fprintf(fp, "# WEIGHTS\n");
```

```
   /* to each node */
   wp = wt;
   for (i = 0; i < nn; i++, wp++){
      fprintf(fp, "# TO NODE %d\n",i+1);
      w = *wp;
      /* from each bias, input, and node */
      for (j = 0; j < nt; j++,w++){
         fprintf(fp,"%f\n",*w);
      }
   }

   fflush(fp);
   fclose(fp);

   if (avq_flag) {
      if (!(fp = fopen(means_file, "w+"))) {
         perror(means_file);
         exit(1);
      }

      WriteCodebook(fp, avq_codebook, codebook_size);

      fflush(fp);
      fclose(fp);
   }

   return;
}

load_wts()
{

   FILE  *fp;
   FILE  *fopen();

   register int   i;
   register int   j;

   register float *w;
   register float **wp;

   int   tmp;

   char  mode[10],
      weights_file[128],
      means_file[128];

   cluster  *saved_book;

   sprintf(weights_file, "%s.wts", loadfile);
   sprintf(means_file, "%s.means", loadfile);

   if ((fp=fopen(weights_file, "r")) == NULL) {
      perror(weights_file);
      exit(1);
   }
   fscanf(fp, "NETWORK CONFIGURED BY %s\n", mode);
   if (strcmp(mode, "TLEARN") != 0) {
      printf("Saved weights not for tlearn-configured network\n");
      exit(1);
   }
   fscanf(fp, "# weights after %ld sweeps\n", &tsweeps);
   fscanf(fp, "# WEIGHTS\n");

   /* to each of nn nodes */
   wp = wt;
   for (i = 0; i < nn; i++, wp++){
      fscanf(fp, "# TO NODE %d\n",&tmp);
      w = *wp;
      /* from each bias, input, and node */
      for (j = 0; j < nt; j++, w++){
         fscanf(fp,"%f\n",w);
      }
   }

   fclose(fp);

   if (avq_flag) {
      if (!(fp = fopen(means_file, "r"))) {
         perror(means_file);
         exit(1);
```

```
      }

      saved_book = ClusterReadFromFile(fp);
      avq_codebook = CodebookNewFromCluster(saved_book, &codebook_size);

      fclose(fp);
      saved_book = ClusterDestroy(saved_book);
   }

   return;
}
```

arrays.c

```
/* make_arrays() - malloc space for arrays */

#include <stdio.h>

#include "vector_utils.h"
#define  MAX_CODEBOOK   500

#ifdef ibmpc
extern char  far *malloc();
#else
extern char *malloc();
#endif
extern   int   nn;      /* number of nodes */
extern   int   ni;      /* number of inputs */
extern   int   no;      /* number of outputs */
extern   int   nt;      /* nn + ni + 1 */

struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight limits */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};

extern   struct   cf **cinfo; /* (nn x nt) connection info */
extern   struct   nf *ninfo;     /* (nn) node activation function info */

extern   int   *outputs;   /* (no) indices of output nodes */
extern   int   *selects;   /* (nn+1) nodes selected for probe printout */
extern   int   *clustnodes;   /* (nn+1) nodes for avq clustering */
extern   int   *contxtnodes;  /* (nn+1) nodes specified as context */
extern   int   *linput; /* (ni) localist input array */
extern   int   *cluster;   /* (nn+1) nodes for avq */
extern   int   *context;   /* (nn+1) nodes specified as context nodes */

extern   vector   **avq_codebook;   /* current codebook */


extern   float *znew;      /* (nt) inputs and activations at time t+1 */
extern   float *zold;      /* (nt) inputs and activations at time t */
extern   float *zmem;      /* (nt) inputs and activations at time t */
extern   float **wt;    /* (nn x nt) weight TO node i FROM node j*/
extern   float **dwt;      /* (nn x nt) delta weight at time t */
extern   float **winc;     /* (nn x nt) accumulated weight increment*/
extern   float *target; /* (no) output target values */
extern   float *error;     /* (nn) error = (output - target) values */
extern   float ***pnew; /* (nn x nt x nn) p-variable at time t+1 */
extern   float ***pold; /* (nn x nt x nn) p-variable at time t */


make_arrays()
{

   int   i;
   int   j;

   struct   cf *ci;
   struct   nf *n;
```

```
zold = (float *) malloc(nt * sizeof(float));
if (zold == NULL){
   perror("zold malloc failed");
   exit(1);
}
zmem = (float *) malloc(nt * sizeof(float));
if (zmem == NULL){
   perror("zmem malloc failed");
   exit(1);
}
znew = (float *) malloc(nt * sizeof(float));
if (znew == NULL){
   perror("znew malloc failed");
   exit(1);
}
target = (float *) malloc(no * sizeof(float));
if (target == NULL){
   perror("target malloc failed");
   exit(1);
}
error = (float *) malloc(nn * sizeof(float));
if (error == NULL){
   perror("error malloc failed");
   exit(1);
}
selects = (int *) malloc(nt * sizeof(int));
if (selects == NULL){
   perror("selects malloc failed");
   exit(1);
}
clustnodes = (int *) malloc(nt * sizeof(int));
if (clustnodes == NULL){
   perror("clustnodes malloc failed");
   exit(1);
}
contxtnodes = (int *) malloc(nt * sizeof(int));
if (contxtnodes == NULL){
   perror("contxtnodes malloc failed");
   exit(1);
}
outputs = (int *) malloc(no * sizeof(int));
if (outputs == NULL){
   perror("outputs malloc failed");
   exit(1);
}
linput = (int *) malloc(ni * sizeof(int));
if (linput == NULL){
   perror("linput malloc failed");
   exit(1);
}

wt = (float **) malloc(nn * sizeof(float *));
if (wt == NULL){
   perror("wt malloc failed");
   exit(1);
}
for (i = 0; i < nn; i++){
   *(wt + i) = (float *) malloc(nt * sizeof(float));
   if (*(wt + i) == NULL){
      perror("wt malloc failed");
      exit(1);
   }
}

dwt = (float **) malloc(nn * sizeof(float *));
if (dwt == NULL){
   perror("dwt malloc failed");
   exit(1);
}
for (i = 0; i < nn; i++){
   *(dwt + i) = (float *) malloc(nt * sizeof(float));
   if (*(dwt + i) == NULL){
      perror("dwt malloc failed");
      exit(1);
   }
}

winc = (float **) malloc(nn * sizeof(float *));
if (winc == NULL){
   perror("winc malloc failed");
   exit(1);
```

```
   }
   for (i = 0; i < nn; i++){
      *(winc + i) = (float *) malloc(nt * sizeof(float));
      if (*(winc + i) == NULL){
         perror("winc malloc failed");
         exit(1);
      }
   }

   cinfo = (struct cf **) malloc(nn * sizeof(struct cf *));
   if (cinfo == NULL){
      perror("cinfo malloc failed");
      exit(1);
   }
   for (i = 0; i < nn; i++){
      *(cinfo + i) = (struct cf *) malloc(nt * sizeof(struct cf));
      if (*(cinfo + i) == NULL){
         perror("cinfo malloc failed");
         exit(1);
      }
   }

   ninfo = (struct nf *) malloc(nn * sizeof(struct nf));
   if (ninfo == NULL){
      perror("ninfo malloc failed");
      exit(1);
   }

   n = ninfo;
   for (i = 0; i < nn; i++, n++){
      n->func = 0;
      n->dela = 0;
      n->targ = 0;
      ci = *(cinfo + i);
      for (j = 0; j < nt; j++, ci++){
         ci->con = 0;
         ci->fix = 0;
         ci->num = 0;
         ci->lim = 0;
         ci->min = 0.;
         ci->max = 0.;
      }
   }

}

make_parrays()
{

   int   i;
   int   j;

   pold = (float ***) malloc(nn * sizeof(float **));
   if (pold == NULL){
      perror("pold malloc failed");
      exit(1);
   }
   for (i = 0; i < nn; i++){
      *(pold + i) = (float **) malloc(nt * sizeof(float *));
      if (*(pold + i) == NULL){
         perror("pold malloc failed");
         exit(1);
      }
      for (j = 0; j < nt; j++){
         *(*(pold + i) + j) = (float *) malloc(nn * sizeof(float));
         if (*(*(pold + i) + j) == NULL){
            perror("pold malloc failed");
            exit(1);
         }
      }
   }

   pnew = (float ***) malloc(nn * sizeof(float **));
   if (pnew == NULL){
      perror("pnew malloc failed");
      exit(1);
   }
   for (i = 0; i < nn; i++){
      *(pnew + i) = (float **) malloc(nt * sizeof(float *));
      if (*(pnew + i) == NULL){
         perror("pnew malloc failed");
         exit(1);
```

```
      }
      for (j = 0; j < nt; j++){
         *(*(pnew + i) + j) = (float *) malloc(nn * sizeof(float));
         if (*(*(pnew + i) + j) == NULL){
            perror("pnew malloc failed");
            exit(1);
         }
      }
   }

}
```

parse.c

```
#include <stdio.h>
#ifdef ibmpc
#include "strings.h"
#else
#include <string.h>
#endif
#include <math.h>

char  pbuf[4096]; /* retains all strings parsed so far */
char  nbuf[256];  /* shows string in get_nums */
int   nbp;

double   atof();

#ifdef ibmpc
extern char  far *malloc();
#else
extern char  *malloc();
#endif
extern   int   nn;      /* number of nodes */
extern   int   ni;      /* number of inputs */
extern   int   no;      /* number of outputs */
extern   int   nt;      /* nn + ni + 1 */
extern   int   np;      /* ni + 1 */
extern   int   nhc;     /* number of hidden/context units */

extern   struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight limits */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

extern   struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};

extern   struct   cf **cinfo; /* (nn x nt) connection info */

extern   struct   nf *ninfo;     /* (nn) node info */

extern   FILE  *cfp;

extern   int   *outputs;   /* (no) indices of output nodes */
extern   int   *selects;   /* (nn+1) nodes selected for probe printout */
extern   int   *clustnodes;   /* (nn+1) nodes to perfoem avq on */
extern   int   *contxtnodes;  /* (nn+1) nodes specified as context */

extern   int   ngroups; /* number of groups */
extern   int   limits;     /* flag for limited weights */

extern   float weight_limit;  /* bound for random weight init */

get_nodes()
{
   int   i;

   char  buf[256];
   char  tmp[256];

   /* read nn, ni, no */
   nn = ni = no = -1;
   get_str(cfp,buf,"\n");
```

```
   /* first line must be "NODES:" */
   if (strcmp(buf, "NODES:") != 0){
      fprintf(stderr,".cf file must begin with NODES:\n");
      exit(1);
   }
   /* next three lines must specify nn, ni, and no in any order */
   for (i = 0; i < 3; i++){
      get_str(cfp,buf," ");
      get_str(cfp,tmp," ");
      if (tmp[0] != '=')
         parse_err();
      get_str(cfp,tmp,"\n");
      if (strcmp(buf, "nodes") == 0)
         nn = atoi(tmp);
      if (strcmp(buf, "inputs") == 0)
         ni = atoi(tmp);
      if (strcmp(buf, "outputs") == 0)
         no = atoi(tmp);
   }
   if ((nn < 1) || (ni < 0) || (no < 0) || (nn < no)){
      fprintf(stderr,"ERROR: Invalid specification\n\n");
      parse_err();
   }
   nt = 1 + ni + nn;
   np = 1 + ni;
}

get_outputs()
{
   int   i;
   int   j;

   struct   nf *n;

   char  buf[256];

   get_str(cfp,buf," ");
   /* next line must specify outputs */
   if (strcmp(buf, "output") != 0)
      parse_err();
   get_str(cfp,buf," ");
   if (strcmp(buf, "node") != 0){
      if (strcmp(buf, "nodes") != 0)
         parse_err();
   }
   get_str(cfp,buf," ");
   if (strcmp(buf, "are") != 0){
      if (strcmp(buf, "is") != 0)
      parse_err();
   }
   get_str(cfp,buf,"\n");
   get_nums(buf,nn+ni,ni,selects);
   if (selects[0] == 1){
      fprintf(stderr,"Node 0 cannot be an output\n");
      exit(1);
   }
   for (i = 1; i <= ni; i++){
      if (selects[i] == 1){
         fprintf(stderr,"An input cannot be an output\n");
         exit(1);
      }
   }
   n = ninfo;
   for (i = ni+1, j = -1; i <= nn+ni; i++, n++){
      if (selects[i] > 0){
         if (++j < no){
            outputs[j] = i-ni;
            n->targ = 1;
         }
      }
   }
   if (++j != no){
      fprintf(stderr,"Expecting %d outputs, found %d\n",no,j);
      exit(1);
   }

}

get_connections()
{
   int   i;
   int   j;
```

```
int   k;

struct   cf *ci;

int   gn;

float min;
float max;

char  buf[256];

int   *tmp;
int   *iselects;

/* malloc space for iselects */
iselects = (int *) malloc(nt * sizeof(int));
if (iselects == NULL){
   perror("iselects malloc failed");
   exit(1);
}

get_str(cfp,buf,"\n");
/* next line must be "CONNECTIONS:" */
if (strcmp(buf, "CONNECTIONS:") != 0)
   parse_err();

get_str(cfp,buf," ");

/* next line must be "groups = #" */
if (strcmp(buf, "groups") != 0)
   parse_err();
get_str(cfp,buf," ");
if (buf[0] != '=')
   parse_err();
get_str(cfp,buf,"\n");
ngroups = atoi(buf);

/* malloc space for tmp */
tmp = (int *) malloc((ngroups+1) * sizeof(int));
if (tmp == NULL){
   perror("tmp malloc failed");
   exit(1);
}

get_str(cfp,buf," ");
while (strcmp(buf, "SPECIAL:") != 0){
   /* a group is identified */
   if (strcmp(buf,"group") == 0){
      get_str(cfp,buf," ");
      get_nums(buf,ngroups,0,tmp);
      get_str(cfp,buf," ");
      if (buf[0] != '=')
         parse_err();
      get_str(cfp,buf," ");
      /* group * = fixed */
      if (strcmp(buf,"fixed") == 0){
         for (i = 0; i < nn; i++){
            ci = *(cinfo + i);
            for (j = 0; j < nt; j++, ci++){
               if (tmp[ci->num])
                  ci->fix = 1;
            }
         }
      }
      /* group * = wmin & wmax */
      else {
         min = (float) atof(buf);
         get_str(cfp,buf," ");
         if (buf[0] != '&')
            parse_err();
         get_str(cfp,buf," ");
         max = (float) atof(buf);
         if (max < min){
            fprintf(stderr,"ERROR: %g < %g\n\n",max,min);
            parse_err();
         }
         for (i = 0; i < nn; i++){
            ci = *(cinfo + i);
            for (j = 0; j < nt; j++, ci++){
               if (tmp[ci->num]){
                  limits = 1;
                  ci->lim = 1;
```

```
                ci->min = min;
                ci->max = max;
             }
          }
       }
    }
    strcat(pbuf,"\n");
    get_str(cfp,buf," ");
 }
 /* a connection is specified */
 else {
    get_nums(buf,nn+ni,ni,selects);
    if (selects[0]){
       fprintf(stderr,"Connecting TO a bias\n\n");
       parse_err();
    }
    for (i = 1; i <= ni; i++){
       if (selects[i]){
          fprintf(stderr,"Connecting TO an input\n\n");
          parse_err();
       }
    }
    get_str(cfp,buf," ");
    if (strcmp(buf,"from") != 0)
       parse_err();
    get_str(cfp,buf," ");
    get_nums(buf,nn+ni,ni,iselects);
    for (i = 0; i < nn; i++){
       ci = *(cinfo + i);
       for (j = 0; j < nt; j++, ci++){
          if ((selects[i+ni+1]) && (iselects[j]))
             ci->con += 1;
       }
    }
    strcat(pbuf,"\t");
    get_str(cfp,buf," ");
    if (buf[0] == '='){
       get_str(cfp,buf," ");
       /* connection  = fixed */
       if (strcmp(buf,"fixed") == 0){
          for (i = 0; i < nn; i++){
             ci = *(cinfo + i);
             for (j = 0; j < nt; j++, ci++){
                if ((selects[i+ni+1]) &&
                   (iselects[j]))
                   ci->fix = 1;
             }
          }
       }
       else {
           /* connection  = group # */
           if (strcmp(buf,"group") == 0){
          get_str(cfp,buf," ");
          gn = atoi(buf);
          for (i = 0; i < nn; i++){
              ci = *(cinfo + i);
              for (j = 0; j < nt; j++, ci++){
             if ((selects[i+ni+1]) &&
                (iselects[j]))
                ci->num = gn;
              }
          }
           }
           /* connection  = min & max */
           else {
          min = (float) atof(buf);
          get_str(cfp,buf," ");
          if (buf[0] != '&')
             parse_err();
          get_str(cfp,buf," ");
          max = (float) atof(buf);
          if (max < min){
             fprintf(stderr,"ERROR: %g < %g\n\n",max,min);
             parse_err();
          }
          for (i = 0; i < nn; i++){
             ci = *(cinfo + i);
             for (j = 0; j < nt; j++, ci++){
                if ((selects[i+ni+1]) &&
                   (iselects[j])){
                   limits = 1;
                   ci->lim = 1;
```

```
                        ci->min = min;
                        ci->max = max;
                     }
                  }
               }
                }
            }
            get_str(cfp,buf,"\t");
            if (strcmp(buf,"fixed") == 0){
               for (i = 0; i < nn; i++){
                  ci = *(cinfo + i);
                  for (j = 0; j < nt; j++, ci++){
                     if ((selects[i+ni+1]) &&
                        (iselects[j]))
                        ci->fix = 1;
                  }
               }
               get_str(cfp,buf,"\t");
            }
            if (strcmp(buf,"one-to-one") == 0){
               for (k = 0; k < nt; k++){
                  if (iselects[k])
                     break;
               }
               for (i = 0; i < nn; i++){
                  ci = *(cinfo + i);
                  for (j = 0; j < nt; j++, ci++){
                     if ((selects[i+ni+1]) &&
                        (iselects[j])){
                        if (ci->con == 1){
                           ci->con = 0;
                           ci->fix = 0;
                           ci->lim = 0;
                        }
                        else
                           ci->con -= 1;
                     }
                  }
                  if (selects[i+np]){
                     ci = *(cinfo + i) + k++;
                     ci->con = 1;
                     ci->fix = 1;
                     ci->lim = 1;
                  }
               }
               get_str(cfp,buf,"\n");
            }
         }
      }
   }
/*
   for (i = 0; i < nn; i++){
      ci = *(cinfo + i);
      for (j = 0; j < nt; j++, ci++){
         fprintf(stderr,"i: %d  j: %d  c: %d  f: %d  g: %d\n",
               i,j,ci->con,ci->fix,ci->num);
      }
   }
*/

}

get_special()
{
   char  buf[256];
   char  tmp[256];

   int   i;

   int   *iselects;

   struct   nf *n;

   /* malloc space for iselects */
   iselects = (int *) malloc(nt * sizeof(int));
   if (iselects == NULL){
      perror("iselects malloc failed");
      exit(1);
   }


   while (fscanf(cfp,"%s",buf) != EOF){
```

```
      strcat(pbuf,buf);
      strcat(pbuf," ");
      get_str(cfp,tmp," ");
      if (tmp[0] != '=')
         parse_err();
      get_str(cfp,tmp,"\n");

      if (strcmp(buf, "cluster") == 0) {
         get_nums(tmp, nn, 0, clustnodes);
         nhc = 1;
      }
      if (strcmp(buf, "context") == 0){
         get_nums(tmp, nn, 0, contxtnodes);
         nhc = 1;
      }

      if (strcmp(buf,"weight_limit") == 0)
         weight_limit = (float) atof(tmp);
      if (strcmp(buf,"selected") == 0){
         get_nums(tmp,nn,0,selects);
      }
      if (strcmp(buf,"linear") == 0){
         get_nums(tmp,nn,0,iselects);
         n = ninfo;
         for (i = 1; i <= nn; i++, n++){
            if (iselects[i])
               n->func = 2;
         }
      }
      if (strcmp(buf,"bipolar") == 0){
         get_nums(tmp,nn,0,iselects);
         n = ninfo;
         for (i = 1; i <= nn; i++, n++){
            if (iselects[i])
               n->func = 1;
         }
      }
      if (strcmp(buf,"binary") == 0){
         get_nums(tmp,nn,0,iselects);
         n = ninfo;
         for (i = 1; i <= nn; i++, n++){
            if (iselects[i])
               n->func = 3;
         }
      }
      if (strcmp(buf,"delayed") == 0){
         get_nums(tmp,nn,0,iselects);
         n = ninfo;
         for (i = 1; i <= nn; i++, n++){
            if (iselects[i])
               n->dela = 1;
         }
      }
   }

}

get_str(fp,buf,str)
   FILE  *fp;
   char  *buf;
   char  *str;
{
   if (fscanf(fp,"%s",buf) == EOF){
      fprintf(stderr,"Premature EOF detected.\n\n");
      parse_err();
   }
   strcat(pbuf,buf);
   strcat(pbuf,str);
}

get_nums(str,nv,offset,vec)
   char  *str;
   int   nv;
   int   offset;
   int   *vec;
{
   int   c, i, j, l, k, m, n;
   int   dash;
   int   input;

   char  tmp[256];
```

```
dash = 0;
input = 0;
l = strlen(str);
nbp = 0;
for (i = 0; i <= nv; i++)
   vec[i] = 0;
for (i = 0, j = 0; i < l; j++, i++){
   c = str[i];
   switch (c) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':   nbuf[nbp++] = tmp[j] = str[i];
            break;
      case 'i':   input++;
            j--;
            nbuf[nbp++] = str[i];
            break;
      case '-':   if (j == 0)
               parse_err();
            tmp[j] = '\0';
            j = -1;
            nbuf[nbp++] = str[i];
            m = atoi(tmp);
            dash = 1;
            break;
      case ',':   if (j == 0)
               parse_err();
            tmp[j] = '\0';
            j = -1;
            nbuf[nbp++] = str[i];
            if (dash){
               n = atoi(tmp);
               if (input == 1)
                  parse_err();
               if (n < m)
                  parse_err();
            }
            else{
               m = atoi(tmp);
               n = m;
            }
            if (input == 0){
               m += offset;
               n += offset;
            }
            if (n > nv){
               fprintf(stderr,"ERROR %d > %d\n",n,nv);
               parse_err();
            }
            if ((input) && (n > offset)){
               fprintf(stderr,"ERROR %d > %d\n",n,offset);
               parse_err();
            }
            for (k = m; k <= n; k++){
               if ((input == 0) && (k == offset))
                  vec[0] = 1;
               else
                  vec[k] = 1;
            }
            input = 0;
            dash = 0;
            break;

      default: parse_err();
   }
}
if (j == 0)
   parse_err();
tmp[j] = '\0';
nbuf[nbp++] = tmp[j];
if (dash){
   n = atoi(tmp);
   if (input == 1){
      fprintf(stderr,"Cannot use dash to connect input and noninput\n\n");
      parse_err();
```

```
      }
      if (n < m){
         fprintf(stderr,"Upper bound must exceed lower\n\n");
         parse_err();
      }
   }
   else{
      m = atoi(tmp);
      n = m;
   }
   if (input == 0){
      m += offset;
      n += offset;
   }
   if (n > nv){
      fprintf(stderr,"ERROR %d > %d\n",n,nv);
      parse_err();
   }
   if ((input) && (n > offset)){
      fprintf(stderr,"ERROR %d > %d\n",n,offset);
      parse_err();
   }
   for (k = m; k <= n; k++){
      if ((input == 0) && (k == offset))
         vec[0] = 1;
      else
         vec[k] = 1;
   }

   nbp = 0;

}

parse_err()
{
   fprintf(stderr,"\nError in .cf file:\n\n");
   fprintf(stderr,"%s\n\n",pbuf);
   fprintf(stderr,"(%s)\n\n",nbuf);
   exit(1);

}
```

activate.c

```
#include <math.h>
#include <stdio.h>


#if defined(ibmpc) || defined(sparc)
#define EXP(m) exp(m)
#else
#define EXP(m) \
   (exp_array[((int) ((m) * exp_mult)) + exp_add])
#endif


extern   int   nn;      /* number of nodes */
extern   int   ni;      /* number of inputs */
extern   int   no;      /* number of outputs */
extern   int   nt;      /* nn + ni + 1 */
extern   int   np;      /* ni + 1 */

extern   struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight limits */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

extern   struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};

extern   struct   cf **cinfo; /* (nn x nt) connection info */

extern   struct   nf *ninfo;  /* (nn) node info */
```

```
extern   int   backprop;   /* flag for back propagation */
extern   int   localist;   /* flag for localist input */
extern   int   teacher; /* flag for feeding back target */

act_nds(aold,amem,anew,awt,local,atarget)
   float *aold;
   float *amem;
   float *anew;
   float **awt;
   int   *local;
   float *atarget;
{

   extern   float *exp_array; /* table look-up for exp function */
   extern   float exp_mult;
   extern   long  exp_add;

   register int   i;
   register int   j;

   register struct   cf *ci;

   register float *w;
   register float *zo;
   register float *zn;
   register float *zp;

   register float **wp;

   register struct   cf **cp;
   register struct   nf *n;
   register struct   nf *on;

   register int   *l;
   register float *t;
   register int   tcnt;

   /* for each of nn nodes: update activations */
   if (backprop == 0){
      zo = aold + np;
      zn = anew + np;
      for (i = 0; i < nn; i++, zo++, zn++){
         *zo = *zn;
      }
   }
   /* remember current aold in amem */
   zo = aold + np;
   zn = amem + np;
   for (i = 0; i < nn; i++, zo++, zn++){
      *zn = *zo;
   }

   /* for each of nn nodes: update net inputs */
   n = ninfo;
   cp = cinfo;
   wp = awt;
   zn = anew + np;
   zp = aold + np;
   t = atarget;
   tcnt = 0;
   for (i = 0; i < nn; i++, zn++, n++, cp++, wp++, zp++){
      if (localist){
         ci = *cp + np;
         w = *wp + np;
         zo = aold + np;
         *zn = **wp;
         l = local;
         while (*l != 0){
            *zn += *(*wp + *l++);
         }
         if (teacher){
            on = ninfo;
            for (j = 0; j < nn; j++, w++, zo++, ci++, on++){
               if (ci->con){
                  if (on->targ){
                     if (*t == -9999.) /* don't care */
                        *zn += *w * *zo;
                     else
                        *zn += *w * *t++;
                     if (++tcnt > no){
                        fprintf(stderr,"WHOA!   -t  flag  requires  each  output  feeding  exactly  one
node\n");
```

```
                        exit(1);
                     }
                  }
                  else
                     *zn += *w * *zo;
               }
            }
         }
         else {
            for (j = 0; j < nn; j++, w++, zo++, ci++){
               if (ci->con)
                  *zn += *w * *zo;
            }
         }
         /* apply activation function  */
         /* 0 = default    */
         /* 1 = bipolar    */
         /* 2 = linear     */
         /* 3 = binary     */
         if (n->func != 2){
            if (*zn > 10.)
               *zn = 10.;
            else if (*zn < -10.)
               *zn = -10.;
         }
         if (n->func < 2)
            *zn = 1.0 / (1.0 + EXP(0.0 - *zn));
         if (n->func == 1)
            *zn = 2. * *zn - 1.;
         else if (n->func == 3){
            if (*zn > 0.)
               *zn = 1.;
            else
               *zn = -1.;
         }
         /* if no-delay, then update immediately */
         if (n->dela == 0)
            *zp = *zn;
      }
      else {
         ci = *cp;
         w = *wp;
         zo = aold;
         *zn = 0.;
         /* collect excitation */
         if (teacher){
            for (j = 0; j <= ni; j++, w++, zo++, ci++){
               if (ci->con)
                  *zn += *w * *zo;
            }
            on = ninfo;
            for (j = 0; j < nn; j++, w++, zo++, ci++, on++){
               if (ci->con){
                  if (on->targ){
                     if (*t == -9999.)
                        *zn += *w * *zo;
                     else
                        *zn += *w * *t++;
                     if (++tcnt > no){
                        fprintf(stderr,"WHOA!   -t  flag  requires  each  output  feeding  exactly  one
node\n");
                        exit(1);
                     }
                  }
                  else
                     *zn += *w * *zo;
               }
            }
         }
         else {
            for (j = 0; j < nt; j++, w++, zo++, ci++){
               if (ci->con)
                  *zn += *w * *zo;
            }
         }
         /* apply activation function  */
         /* 0 = default    */
         /* 1 = bipolar    */
         /* 2 = linear     */
         /* 3 = binary     */
         if (n->func != 2){
            if (*zn > 10.)
```

```
            *zn = 10.;
         else if (*zn < -10.)
            *zn = -10.;
      }
      if (n->func < 2)
         *zn = 1.0 / (1.0 + EXP(0.0 - *zn));
      if (n->func == 1)
         *zn = 2. * *zn - 1.;
      else if (n->func == 3){
         if (*zn > 0.)
            *zn = 1.;
         else
            *zn = -1.;
      }
      /* if no-delay, then update immediately */
      if (n->dela == 0)
         *zp = *zn;
   }
 }

}
```

compute.c

```
#include <math.h>
#include <stdio.h>

#ifdef ibmpc
extern char  far *malloc();
#else
extern char  *malloc();
#endif


extern   int   nn;      /* number of nodes */
extern   int   ni;      /* number of inputs */
extern   int   no;      /* number of outputs */
extern   int   nt;      /* nn + ni + 1 */
extern   int   np;      /* ni + 1 */
extern   int   ce;      /* cross-entropy flag */

struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight limits */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

extern   struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};

extern   struct   cf **cinfo; /* (nn x nt) connection info */
extern   struct   nf *ninfo;     /* (nn) node activation function info */

extern   int   *outputs;   /* (no) indices of output nodes */

extern   int   localist;   /* flag for localist input */

comp_errors(aold,atarget,aerror,e,ce_e)
   float *aold;
   float *atarget;
   float *aerror;
   float *e;
   float *ce_e;
{
   extern   int ce;

   register int   i;
   register int   j;
   register float *ta;
   register float *te;
   register float *ce_te;
   register float *ee;
   register int   *op;
```

```
   static   float *terror = 0;
   static   float *ce_terror = 0;

   if (terror == 0){
      /* malloc space for local copy of error info */
      terror = (float *) malloc(no * sizeof(float));
      if (terror == NULL){
         perror("terror malloc failed");
         exit(1);
      }
   }
   if (ce_terror == 0){
      /* malloc space for local copy of cross-entropy info */
      ce_terror = (float *) malloc(no * sizeof(float));
      if (ce_terror == NULL){
         perror("ce_terror malloc failed");
         exit(1);
      }
   }


   te = terror;
   ce_te = ce_terror;
   ta = atarget;
   op = outputs;
   for (i = 0; i < no; i++, te++, ce_te++, ta++, op++){
      if (*ta != -9999.0) {
         *te = *(aold + ni + *op) - *ta;
         /*
          * if collecting cross-entropy statistics;
          */
         if (ce == 2) {
             *ce_te = *ta * log(*(aold+ni+ *op))/log(2.0) +
                (1- *ta) * log(1- *(aold+ni+ *op))/log(2.0);
         }
      } else {
         *te = 0.;
      }
      *e += *te * *te;  /* cumulative ss error */
      *ce_e += *ce_te;  /* cumulate cross-entropy error */
   }
   ee = aerror;
   for (i = 1; i <= nn; i++, ee++){
      *ee = 0.;
      te = terror;
      op = outputs;
      for (j = 0; j < no; j++, te++, op++){
         if (*op == i){
            *ee = *te;
            break;
         }
      }
   }
}


comp_deltas(apold,apnew,awt,adwt,aold,anew,aerror)
   float ***apold;
   float ***apnew;
   float **awt;
   float **adwt;
   float *aold;
   float *anew;
   float *aerror;
{
   register int   i;
   register int   j;
   register int   k;
   register int   l;

   register struct   cf **cp;

   register struct   cf *ci;
   register struct   nf *n;

   register float **wp;
   register float *zn;
   register float *pn;
   register float *po;
   register float **pnp;
   register float **pop;
   register float ***pnpp;
```

```
   register float ***popp;
   register float *w;

   register float *sum;

   register float *e;

   float asum;

   /* to each node */
   sum = &asum;
   cp = cinfo;
   pnpp = apnew;
   popp = apold;
   for (i = 0; i < nn; i++, cp++, pnpp++, popp++){
      ci = *cp;
      pnp = *pnpp;
      pop = *popp;
      /* from each bias, input, and node */
      for (j = 0; j < nt; j++, ci++, pnp++, pop++){
         if (ci->con == 0)
            continue;
         pn = *pnp;
         zn = anew + np;
         n = ninfo;
         /* for each node */
         for (k = 0; k < nn; k++, zn++, pn++, n++){
            w = *(awt + k) + np;
            po = *pop;
            if (i == k)
               *sum = *(aold + j);
            else
               *sum = 0.;
            /* from each node */
            for (l = 0; l < nn; l++, w++, po++){
               *sum += *w * *po;
            }
            if (n->func == 0)
               *pn = *zn * (1. - *zn) * *sum;
            else if (n->func == 1)
               *pn = .5 * (1. + *zn)*(1. - *zn) * *sum;
            else if (n->func == 2){
               *pn = *sum;
            }
            if (n->dela == 0)
               *(*(*(apold + i) + j) + k) = *pn;
         }
      }
   }
   /* to each node */
   cp = cinfo;
   wp = adwt;
   pnpp = apnew;
   popp = apold;
   for (i = 0; i < nn; i++, cp++, wp++, pnpp++, popp++){
      w = *wp;
      ci = *cp;
      pnp= *pnpp;
      pop= *popp;
      /* from each bias, input, and node */
      for (j = 0; j < nt; j++, w++, ci++, pnp++, pop++){
         if (ci->con == 0)
            continue;
         e = aerror;
         pn = *pnp;
         po = *pop;
         *sum = 0.;
         /* for each node */
         for (k = 0; k < nn; k++, e++, po++, pn++){
            *sum += *e * *po;
            *po = *pn;
         }
         *w -= *sum;
      }
   }

   return;
}

comp_backprop(awt,adwt,aold,amem,atarget,aerror,local)
   float **awt;
   float **adwt;
```

```
   float *aold;
   float *amem;
   float *atarget;
   float *aerror;
   int   *local;
{
   register int   i;
   register int   j;

   register struct   cf **cp;

   register struct   cf *ci;
   register struct   nf *n;

   register float *sum;

   float **wp;
   float *ee;
   float *e;
   float *w;
   float *z;
   float *oz;
   float *t;

   int   *l;
   int   ns;

   float asum;

   /* compute deltas for output units */
   sum = &asum;
   e = aerror;
   n = ninfo;
   z = aold + np;
   t = atarget;
   for (i = 0; i < nn; i++, e++, n++, z++){
      if (n->targ == 0)
         continue;
      if (n->func == 0) {
         if (ce > 0) {     /* if cross-entropy */
            /*
             * note that the following collapses
             * (t-a) and derivative of slope; we
             * therefore ignore current contents of
             * *e (which is (t-a)) and assign new
             * value, whereas with sse, we multiply *e
             * by deriv. of slope.
             */
            *e = *t - *z;
            /* NOTE: this is a kludge -- only increments
             * target when node is an output node. Do
             * NOT move into for() control statement.
             */
            t++;
         } else {    /* otherwise normal sse-delta */
            *e *= *z * (1. - *z);
         }
      } else if (n->func == 1)
         *e *= .5 * (1. + *z) * (1. - *z);
   }

   n = ninfo + nn - 1;
   z = aold + nt - 1;
   e = aerror + nn - 1;
   /* compute deltas for remaining units */
   for (i = nn - 1; i >= 0; i--, z--, e--, n--){
      if (n->targ == 1)
         continue;
      *sum = 0.;
      /* ee contains a bad address for i = nn-1 */
      ee = aerror + i + 1;
      for (j = i + 1; j < nn; j++, ee++){
         w = *(awt + j) + np + i;
         ci = *(cinfo + j) + np + i;
         if (ci->con)
            *sum += *w * *ee;
      }
      if (n->func == 0)
         *e = *sum * *z * (1. - *z);
      else if (n->func == 1)
         *e = *sum * .5 * (1. + *z) * (1. - *z);
      else if (n->func == 2){
```

```
         *e = *sum;
      }
      else if (n->func == 3)
         *e = 0.;
   }

   /* compute weight changes for all connections */

   /* to each node */
   e = aerror;
   cp = cinfo;
   wp = adwt;
   for (i = 0; i < nn; i++, e++, cp++, wp++){
      if (localist){
         if (ce > 0){
            if ((*cp)->con)
               **wp += *e;
         }
         else {
            if ((*cp)->con)
               **wp -= *e;
         }
         l = local;
         while (*l != 0){
            if (ce > 0){
               if ((*cp + *l)->con)
                  *(*wp + *l) += *e;
            }
            else {
               if ((*cp + *l)->con)
                  *(*wp + *l) -= *e;
            }
            l++;
         }
         w = *wp + np;
         ci = *cp + np;
         z = aold + np;
         oz = amem + np;
         /* from each node */
         /* loop is broken into two parts:
              (1) connections from nodes of lower node-number
              (2) connections from nodes of = or > node-number
            the latter case requires use of old z values */
         if (ce > 0){
            for (j = 0; j < i; j++, w++, ci++, z++, oz++){
               if (ci->con)
                  *w += *z * *e;
            }
            for (j = i; j < nn; j++, w++, ci++, z++, oz++){
               if (ci->con)
                  *w += *oz * *e;
            }
         }
         else {
            for (j = 0; j < i; j++, w++, ci++, z++, oz++){
               if (ci->con)
                  *w -= *z * *e;
            }
            for (j = i; j < nn; j++, w++, ci++, z++, oz++){
               if (ci->con)
                  *w -= *oz * *e;
            }
         }
      }
      else {
         w = *wp;
         ci = *cp;
         z = aold;
         oz = amem;
         /* from each bias, input, and node */
         ns = np + i;
         /* loop is broken into two parts:
              (1) connections from nodes of lower node-number
              (2) connections from nodes of = or > node-number
            the latter case requires use of old z values */
         if (ce > 0){
            for (j = 0; j < ns; j++, w++, ci++, z++, oz++){
               if (ci->con)
                  *w += *z * *e;
            }
            for (j = ns; j < nt; j++, w++, ci++, z++, oz++){
               if (ci->con)
```

```
                 *w += *oz * *e;
              }
           }
           else {
              for (j = 0; j < ns; j++, w++, ci++, z++, oz++){
                 if (ci->con)
                    *w -= *z * *e;
              }
              for (j = ns; j < nt; j++, w++, ci++, z++, oz++){
                 if (ci->con)
                    *w -= *oz * *e;
              }
           }
        }
     }

     return;
  }
```

subs.c

```
#include <math.h>
#include <stdio.h>
#ifdef ibmpc
#include <fcntl.h>
#include <sys/types.h>
#else
#include <fcntl.h>
#include <sys/file.h>
#include <sys/types.h>
#endif
#include <sys/stat.h>

#ifdef ibmpc
extern char  far *malloc();
#define random() rand()
#define srandom(x) srand(x)
#else
extern char  *malloc();
#endif

#ifndef  EXP_LOCATION
#error   You must define EXP_LOCATION on the command line
#endif

float *exp_array; /* table look-up for exp function */
float exp_mult;
long  exp_add;

extern   int   nn;      /* number of nodes */
extern   int   ni;      /* number of inputs */
extern   int   no;      /* number of outputs */
extern   int   nt;      /* nn + ni + 1 */

extern   int   ce;      /* cross-entropy flag */

extern   int   *outputs;   /* (no) indices of output nodes */
extern   int   *selects;   /* (nn+1) nodes selected for probe printout */

extern   char  root[128];  /* root filename for .cf, .data, .teach, etc.*/

extern   long  sweep;      /* current sweep */
extern   long  err_sweeps; /* number of sweeps for cumulative error */
extern   long  report;     /* report error every report sweeps */
extern   float avq_err; /* error for avq modification */

extern   float criterion;  /* exit program when rms error < criterion */


float rans(w)
   float w;
{
/* extern   long random(); */
#ifdef ibmpc
   static   float max = 32767.0;
#else
   static   float max = 2147483647.0;
#endif

   return (((float)random() / max) * 2*w) - w;
```

```
}

exp_init()
{
   struct   stat statb;
   int   fd;

   fd = open(EXP_LOCATION, O_RDONLY, 0);

   if (fd < 0) {
      perror("exp_table");
      exit(1);
   }
   fstat(fd, &statb);
   exp_add = (statb.st_size / sizeof(float)) / 2;
   exp_mult = (float) (exp_add / 16);
   exp_array = (float *) malloc(statb.st_size);
   if (read(fd, exp_array, statb.st_size) != statb.st_size) {
      perror("read exp array");
      exit(1);
   }
}

print_nodes(aold)
   float *aold;
{
   int   i;

   for (i = 1; i <= nn; i++){
      if (selects[i])
         fprintf(stdout,"%7.3f\t",aold[ni+i]);
   }
   fprintf(stdout,"\n");

}

print_output(aold)
   float *aold;
{
   int   i;

   for (i = 0; i < no; i++){
      fprintf(stdout,"%7.3f\t",aold[ni+outputs[i]]);
   }
   fprintf(stdout,"\n");

}

print_error(e)
   float *e;
{
   static   int   start = 1;
   static   FILE  *fp;

   FILE  *fopen();
   char  file[128];

   if (start){
      start = 0;
      sprintf(file, "%s.err", root);
      fp = fopen(file, "w");
      if (fp == NULL) {
         perror("ERROR: Can't open .err file");
         exit(1);
      }
   }

   if (ce != 2) {
      /* report rms error */
      *e = sqrt(*e / report);
   } else if (ce == 2) {
      /* report cross-entropy */
      *e = *e / report;
   }
   fprintf(fp,"%g\n",*e);
   fflush(fp);
   if (ce == 0) {
      if (*e < criterion){
         sweep += 1;
         save_wts();
            exit(0);
      }
```

```
   }
   *e = 0.;
   err_sweeps = 0;
}

reset_network(aold,anew,apold,apnew)
   float *aold;
   float *anew;
   float ***apold;
   float ***apnew;
{
   register int   i, j, k;

   register float *pn;
   register float *po;
   register float **pnp;
   register float **pop;
   register float ***pnpp;
   register float ***popp;

   register float *zn;
   register float *zo;

   zn = anew + 1;
   zo = aold + 1;
   for (i = 1; i < nt; i++, zn++, zo++)
      *zn = *zo = 0.;

   popp = apold;
   pnpp = apnew;
   for (i = 0; i < nn; i++, popp++, pnpp++){
      pop = *popp;
      pnp = *pnpp;
      for (j = 0; j < nt; j++, pop++, pnp++){
         po = *pop;
         pn = *pnp;
         for (k = 0; k < nn; k++, po++, pn++){
            *po = 0.;
            *pn = 0.;
         }
      }
   }

   return;
}

reset_bp_net(aold,anew)
   float *aold;
   float *anew;
{
   register int   i;

   register float *zn;
   register float *zo;

   zn = anew + 1;
   zo = aold + 1;
   for (i = 1; i < nt; i++, zn++, zo++)
      *zn = *zo = 0.;

   return;
}
```

update.c

```
#include <stdio.h>

#ifdef ibmpc
#define random() rand()
#define srandom(x) srand(x)
#endif

double   atof();

#ifdef ibmpc
extern char  far *malloc();
#else
extern char  *malloc();
#endif
extern   int   nn;      /* number of nodes */
extern   int   ni;      /* number of inputs */
```

```
extern   int   no;      /* number of outputs */
extern   int   nt;      /* nn + ni + 1 */
extern   int   np;      /* ni + 1 */

extern   struct   cf {
   int   con;  /* connection flag */
   int   fix;  /* fixed-weight flag */
   int   num;  /* group number */
   int   lim;  /* weight limits */
   float min;  /* weight minimum */
   float max;  /* weight maximum */
};

extern   struct   nf {
   int   func; /* activation function type */
   int   dela; /* delay flag */
   int   targ; /* target flag */
};

extern   struct   cf **cinfo; /* (nn x nt) connection info */

extern   int   ngroups; /* number of groups */

extern   char  root[128];  /* root filename for .data, .teach, etc. files */

extern   float rate;    /* learning rate */
extern   float momentum;   /* momentum */

extern   int   randomly;   /* flag for presenting inputs in random order */
extern   int   localist;   /* flag for localist inputs */
extern   int   limits;     /* flag for limited weights */

long  dc = 0;
int   *ldata = 0;


update_inputs(aold,tick,flag,maxtime,local)
   float *aold;
   int   tick;
   int   flag;
   long  *maxtime;
   int   **local;
{
/* extern   long  random(); */

   register int   i;

   int   j;

   long  ii;

   static   long  dn;
   static   long  ds;

   static   float *data = 0;

   int   *idata;
   int   *id;
   int   *lld;

   static   FILE  *fp;

   char  buf[128];
   char  file[128];

   static   float *dm;
   static   int   *ld;

   register float *d;
   register float *zo;


   if ((data == 0) && (ldata == 0)){
      /* get .data file */
      sprintf(file, "%s.data", root);
      fp = fopen(file, "r");
      if (fp == NULL) {
           perror("ERROR: Empty data file");
           exit(1);
      }
      /* determine format of .data file */
      fscanf(fp,"%s",buf);
```

```
    if (strcmp(buf, "localist") == 0){
       localist = 1;
    }
    else if (strcmp(buf, "distributed") != 0){
       perror("ERROR: .data file must be localist or distributed\n");
       exit(1);
    }
    /* determine size of .data file */
    if (fscanf(fp,"%ld",maxtime) != 1){
       perror("ERROR: how many items in .data file?");
       exit(1);
    }
    /* malloc space for data */
    if (localist){
       dn = *maxtime;
       ds = *maxtime * ni;
       ldata = (int *) malloc(ds * sizeof(int));
       if (ldata == NULL){
          perror("ldata malloc failed");
          exit(1);
       }
       idata = (int *) malloc((ni+1) * sizeof(int));
       if (idata == NULL){
          perror("idata malloc failed");
          exit(1);
       }
       /* read data */
       ld = ldata;
       for (ii = 0; ii < dn; ii++){
          fscanf(fp,"%s",buf);
          get_nums(buf,ni,0,idata);
          id = idata + 1;
          lld = ld;
          for (j = 1; j <= ni; j++, id++){
             if (*id)
                *lld++ = j;
          }
          *lld = 0;
          ld += ni;
       }
    }
    else {
       dn = *maxtime;
       ds = *maxtime * ni;
       data = (float *) malloc(ds * sizeof(float));
       if (data == NULL){
          perror("data malloc failed");
          exit(1);
       }
       /* read data */
       d = dm = data;
       for (ii = 0; ii < ds; ii++, d++){
          fscanf(fp,"%s",buf);
          *d = atof(buf);
          if ((*d == 0.) && (buf[0] != '0') && (buf[1] != '0')){
             fprintf(stderr,"error reading .data file on or around line %ld of input\n",ii+1);
             exit(1);
          }
       }
    }
 }


 /* update input (only at major time increments) */
 if (tick == 0){
    /* read next ni inputs from .data file */
    if (localist){
       if (randomly){
          dc = (random() >> 8) % dn;
          if (dc < 0)
             dc = -dc;
          *local = (int *) (ldata + dc * ni);
       }
       else {
          *local = (int *) (ldata + dc * ni);
          if (++dc >= dn)
             dc = 0;
       }
    }
    else {
       if (randomly){
          dc = (random() >> 8) % dn;
```

```
            if (dc < 0)
               dc = -dc;
            d = (float *) (data + dc * ni);
            zo = aold + 1;
            for (i = 0; i < ni; i++, zo++, d++){
               *zo = *d;
            }
         }
         else {
            d = dm;
            zo = aold + 1;
            for (i = 0; i < ni; i++, zo++, d++, dc++){
               if (dc >= ds){
                  dc = 0;
                  d = data;
               }
               *zo = *d;
            }
            dm = d;
         }
      }
   }
   else {
      /* turn off input during extra ticks with -I */
      if (flag){
         zo = aold + 1;
         for (i = 0; i < ni; i++, zo++)
            *zo = 0.;
      }
   }

}

update_targets(atarget,time,tick,flag,maxtime)
   float *atarget;
   long  time;
   int   tick;
   int   flag;
   long  *maxtime;
{
   long  i;

   int   k;

   register int   j;

   register float *ta;
   register float *t;
   register long  *n;
   register float *to;

   static   long  *ntimes;
   static   float *teach;

   static   int   local = 0;  /* flag for localist output */

   static   long  *nm;
   static   float *tm;
   static   long  nc = 0;

   static   long  len;
   static   long  ts;

   static   long  next;    /* next time tag in .teach file */

   static   float *otarget = 0;  /* (no) back-up copy of target values */

   static   FILE  *fp;

   char  buf[128];

   if (otarget == 0){
      /* get .teach file */
      sprintf(buf, "%s.teach", root);
      fp = fopen(buf, "r");
      if (fp == NULL) {
           perror("ERROR: Empty target file");
           exit(1);
      }
      /* malloc space for back-up copy of targets */
      otarget = (float *) malloc(no * sizeof(float));
      if (otarget == NULL){
```

```
         perror("otarget malloc failed");
         exit(1);
      }
      /* determine format of .teach file */
      fscanf(fp,"%s",buf);
      if (strcmp(buf, "localist") == 0){
         local = 1;
      }
      else if (strcmp(buf, "distributed") != 0){
         perror("ERROR: .teach file must be localist or distributed\n");
         exit(1);
      }
      /* determine size of teach array */
      if (fscanf(fp,"%ld",&len) != 1){
         perror("ERROR: how many items in .teach file?");
         exit(1);
      }
      /* malloc space for teach and ntimes buffers */
      ts = len * no;
      teach = (float *) malloc(ts * sizeof(float));
      if (teach == NULL){
         perror("teach malloc failed");
         exit(1);
      }
      ntimes = (long *) malloc(len * sizeof(long));
      if (ntimes == NULL){
         perror("ntimes malloc failed");
         exit(1);
      }
      /* read teach info */
      t = tm = teach;
      n = nm = ntimes;
      for (i = 0; i < len; i++, n++){
         fscanf(fp,"%ld",n);
         if (local){
            fscanf(fp,"%s",buf);
            k = atoi(buf) - 1;
            if (k < 0){
               fprintf(stderr,"error reading .teach file on or around line %ld of input\n",i+1);
               exit(1);
            }
            for (j = 0; j < no; j++, t++){
               if (j == k)
                  *t = 1.;
               else
                  *t = 0.;
            }
         }
         else {
            for (j = 0; j < no; j++, t++){
               fscanf(fp,"%s",buf);
               /* asterick is don't care sign */
               if (buf[0] == '*')
                  *t = -9999.0;
               else {
                  *t = atof(buf);
                  if ((*t == 0.) && (buf[0] != '0') && (buf[1] != '0')){
                     fprintf(stderr,"error   reading   .teach   file   on   or   around   line   %ld   of
input\n",i+1);
                     exit(1);
                  }
               }
            }
         }
      }
   }

   /* check for new target values (only at major time increments) */
   if (tick == 0){
      t = tm;
      n = nm;
      /* restore previous values if destroyed by -T */
      if (flag){
         ta = atarget;
         to = otarget;
         for (j = 0; j < no; j++, ta++, to++)
            *ta = *to;
      }

      /* if inputs are selected randomly, time-tags are
         assumed to run sequentially, and targets are
         selected to match input */
```

```
      if (randomly){
         if (dc >= len){
            perror("ERROR: a target line is required for every input line with -R");
            exit(1);
         }
         ta = atarget;
         t = (float *) (teach + no * dc);
         for (j = 0; j < no; j++, ta++, t++){
            *ta = *t;
         }
         return;
      }

      /* rewind whenever .data begins again at time 0 */
      if (time == 0){
         nc = 0;
         t = teach;
         n = ntimes;
         next = *n;
         ta = atarget;
         for (j = 0; j < no; j++, ta++)
            *ta = -9999.0;
      }
      /* get new target values when time matches next */
      if (time >= next){
         /* read next no targets */
         ta = atarget;
         for (j = 0; j < no; j++, t++, ta++){
            *ta = *t;
         }
         /* final target persists till end of input */
         n++;
         if (++nc >= len)
            next = *maxtime;
         else
            next = *n;
      }
      tm = t;
      nm = n;
      /* remember target values if -T will destroy them */
      if (flag){
         ta = atarget;
         to = otarget;
         for (j = 0; j < no; j++, ta++, to++)
            *to = *ta;
      }
   }
   else {
      /* turn off target during extra ticks with -T */
      if (flag){
         ta = atarget;
         for (j = 0; j < no; j++, ta++)
            *ta = -9999.0;
      }
   }

}

update_reset(time,tick,flag,maxtime,now)
   long  time;
   int   tick;
   int   flag;
   long  *maxtime;
   int   *now;
{
   long  i;

   static   int   start = 1;  /* flag for initialization */
   static   long  next;    /* next time tag in .teach file */

   static   long  *rtimes;

   static   long  *nm;
   static   long  nc = 0;

   static   long  l;

   static   FILE  *fp;

   char  buf[128];
```

```
   *now = 0;

   if (flag == 0)
      return;

   if (start){
      start = 0;
      /* get .reset file */
      sprintf(buf, "%s.reset", root);
      fp = fopen(buf, "r");
      if (fp == NULL) {
           perror("ERROR: Empty reset file");
           exit(1);
      }
      /* determine size of .reset file */
      if (fscanf(fp,"%ld",&l) != 1){
         perror("error reading .reset file");
         exit(1);
      }
      /* malloc space for rtimes buffer */
      rtimes = (long *) malloc(l * sizeof(long));
      if (rtimes == NULL){
         perror("rtimes malloc failed");
         exit(1);
      }
      /* read reset info */
      nm = rtimes;
      for (i = 0; i < l; i++, nm++)
         fscanf(fp,"%ld",nm);
      nm = rtimes;
   }

   /* check for new resets (only at major time increments) */
   if (tick == 0){
      /* rewind whenever .data begins again at time 0 */
      if (time == 0){
         nc = 0;
         nm = rtimes;
         next = *nm;
      }
      if (time >= next){
         *now = 1;
         nm++;
         if (++nc >= l)
            next = *maxtime;
         else
            next = *nm;
      }
   }

}

update_weights(awt,adwt,awinc)
   float **awt;
   float **adwt;
   float **awinc;
{
   register int   i;
   register int   j;

   register struct   cf *ci;

   register float *w;
   register float *dw;
   register float *wi;
   register float **wip;
   register float **wp;
   register float **dwp;

   register struct   cf **cp;

   register int   k;
   register int   n;
   register float *sum;

   float asum;

   /* update weights if they are not fixed */
   sum = &asum;
   cp = cinfo;
   wp = awt;
   dwp = adwt;
```

```
   wip = awinc;
   for (i = 0; i < nn; i++, cp++, wp++, dwp++, wip++){
      ci = *cp;
      w = *wp;
      dw = *dwp;
      wi = *wip;
      for (j = 0; j < nt; j++, dw++, wi++, w++, ci++){
         if ((ci->con) && !(ci->fix)){
            *wi = rate * *dw + momentum * *wi;
            *w += *wi;
            *dw = 0.;
         }
      }
   }
   /* look for weights in the same group and average them together */
   for (k = 1; k <= ngroups; k++){
      *sum = 0.;
      n = 0;
      cp = cinfo;
      wp = awt;
      /* calculate average */
      for (i = 0; i < nn; i++, cp++, wp++){
         ci = *cp;
         w = *wp;
         for (j = 0; j < nt; j++, w++, ci++){
            if (ci->num == k){
               *sum += *w;
               n++;
            }
         }
      }
      if (n > 0)
         *sum /= n;
      /* replace weight with average */
      cp = cinfo;
      wp = awt;
      for (i = 0; i < nn; i++, cp++, wp++){
         ci = *cp;
         w = *wp;
         for (j = 0; j < nt; j++, w++, ci++){
            if (ci->num == k)
               *w = *sum;
         }
      }
   }
   /* look for limited weights and enforce limits */
   if (limits == 0)
      return;
   cp = cinfo;
   wp = awt;
   for (i = 0; i < nn; i++, cp++, wp++){
      ci = *cp;
      w = *wp;
      for (j = 0; j < nt; j++, w++, ci++){
         if (ci->lim){
            if (*w < ci->min)
               *w = ci->min;
            if (*w > ci->max)
               *w = ci->max;
         }
      }
   }

   return;
}
```

9.3 Source Code Modules

9.3.1 StdDefs

Purpose	Provides a set of standard definitions for all modules.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	3rd March, 1998
Modified	12th February, 2001
Module version	1.34

StdDefs.h

```
/* StdDefs.h -- Contains standard definitions for other modules
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 * Date: 3rd March, 1998
 * Modified: 12th February, 2001
 * Version: 1.34
 */

#ifndef  __stddefs_h
#define  __stddefs_h

/* -- Standard macros */

#define  MIN(x, y)   ((x) < (y) ? (x) : (y))
#define  MAX(x, y)   ((x) > (y) ? (x) : (y))
#define  SQR(x)      ((x) * (x))

/*#define   DEBUG                uncomment for debugging */
/*#define   ANSI                 uncomment for ANSI-only code */
/*#define   CHECK_PRECOND        uncomment to check preconditions */

/* -- Hardware type size definitions */

typedef  unsigned char        BYTE;
typedef  signed char          SBYTE;
typedef  unsigned short int   WORD;
typedef  signed short int     SWORD;
typedef  unsigned long int    DWORD;
typedef  signed long int      SDWORD;

/* -- Standard typedefs */

typedef char   *string;

#define  FALSE 0              /******/
#define  TRUE  !FALSE         /* Support for boolean functions */
typedef  int   BOOL;          /******/

/* -- Standard definitions */

#define  MAX_STRING           400

#define  EOL   '\n'
#define  EOS   '\0'
#define  TAB   0x09

#define  MATH_PI  3.1415926

#endif   /* __stddefs_h */

/* --- END of StdDefs.h --- */
```

9.3.2 2darray

Purpose	Creates and destroys easy-to-handle 2D arrays in C.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	7th November, 1999
Modified	16th August, 2000
Module version	0.03
Notes	Creates arrays that can be indexed in C by `array[column][row]`
Other required	StdDefs

2darray.h

```
/* 2darray.h -- Allocate and destroy 2d arrays
 *
 * Author: Dylan Muir
 * Date: 7th November, 1999
 * Modified: 16th August, 2000
 * Version: 0.03
 */

#ifndef  __2darray_h
#define  __2darray_h

#include "stddefs.h"
#include <stdlib.h>

/* -- Function Allocate2DArray
 * Pre: 'array' is a pointer to a pointer to a pointer to char
 *      'xSize' > 0
 *      'ySize' > 0
 *      'elementSize' is the size of each element to allocate
 * Post: ('array' was allocated space for a 2d array of dimensions [xSize x ySize]
 *       && TRUE was returned) ||
 *       (there was a problem with allocation && FALSE was returned)
 * Note: NO initialisation is done!  The content of the array after allocation
 *       is undefined.
 * Note: the array is indexed 'array'[col][row]
 */
bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize);

/* -- Function Deallocate2DArray
 * Pre: 'array' is a pointer to a valid 2D array allocated by Allocate2DArray,
 *       the contents of which have already been destroyed
 *      'ySize' is the size of the first index in 'array'
 * Post: 'array' was deallocated
 * Note: This function does not destroy the contents of the array, as it has no
 *       idea what the contents are.  You must destroy the contents yourself.
 */
void Deallocate2DArray(char ***array, int xSize);

#endif   __2darray_h

/* --- END of 2darray.h --- */
```

2darray.c

```
/* 2darray.c -- 2dimensional array functions
 *
 * SEE 2darray.h for details
 */

#include <stdlib.h>
#include "2darray.h"

bool Allocate2DArray(char ***array, int xSize, int ySize, size_t elementSize)
{
   int     index;

   if (!((*array) = (char **) malloc(ySize * sizeof(void *))))
      return FALSE;

   for (index = 0; index < ySize; index++) {
      if (!((*array)[index] = (char *) malloc(xSize * elementSize))) {
         free(*array);
         return FALSE;
      }
   }

   return TRUE;
}

void Deallocate2DArray(char ***array, int ySize)
{
   int   index;

   for (index = 0; index < ySize; index++)
      free((*array)[index]);
```

```
   free(*array);
   (*array) = NULL;
}

/* --- END of 2darray.c --- */
```

9.3.3 gauss

Purpose	Inverts square 2d matrices using the Gauss-Jordan elimination method. Matrices must be created via the **2darray** module.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	November, 1999
Modified	21st November, 2000
Module version	0.02
Other required modules	StdDefs, 2darray

gauss.h

```
/* Gauss.h -- Provides Gaussian matrix inversion routines
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         (c) QUT MLRC Semester 2 2000
 * Date: November, 1999
 * Modified: 21st November, 2000
 * Version: 0.02
 */

#ifndef  __gauss_h
#define  __gauss_h

/* -- Function GaussJordanInverse
 * Pre: 'matrix' is a square double 2d array allocated by 'Allocate2dArray'
 *      'inverse' is a square double 2d array allocated by 'Allocate2dArray', the
 *       same dimensions as 'matrix'
 *      'size' is the number of rows in 'matrix' and 'inverse'
 * Post: ('matrix' has been inverted and placed into 'inverse' &&
 *        'matrix' is unchanged && TRUE was returned) ||
 *       ('matrix' is singular && 'inverse' is undefined && FALSE was returned)
 */
bool GaussJordanInverse(double **matrix, double **inverse, int size);

#endif   /* __gauss_h */

/* --- END of gauss.h --- */
```

gauss2.c

```
/* Gauss-Jordan implementation */
/* Thanks to John Stearns for inspiration */

#include "2darray.h"
#include "stddefs.h"
#include <math.h>
#ifdef   UNIX
   #include <ieeefp.h>
#endif
#ifdef   WIN32
   #include <float.h>
   #define  FP_PINF  _FPCLASS_PINF
#endif

/* -- Defines */

#define  SWAP(x, y)     {temp = (x); (x) = (y); (y) = temp;}

#define  GJ_SING_SENS   0.00              /* Singular matrix sensitivity */


/* -- Gauss-Jordan inverse */

bool GaussJordanInverse(double **matrix, double **inverse, int size)
{
   double   det,           /* Determinant (product of pivots) */
            temp,          /* Used by SWAP */
```

```
         pivot,         /* Current pivot value */
         factor;        /* Factor of pivot row to multiply by */
   int   ipass,         /* Elimination index (current pivot row) */
         imax,          /* Index of maximum value in pivot column */
         icol,          /* Column index */
         irow;          /* Row index */

   det = 1.0;

   /* Initially inverse is the identity matrix, */
   /* this will be replaced with the inverse    */
   for (irow = 0; irow < size; irow++) {
      for(icol = 0; icol < size; icol++) {
         if (irow == icol)
            inverse[irow][icol] = 1;
         else
            inverse[irow][icol] = 0;
      }
   }

   for (ipass = 0; ipass < size; ipass++) {
      /* Find maximum value in the pivot column */
      imax = ipass;
      for (irow = ipass; irow < size; irow++) {
         if (fabs(matrix[irow][ipass]) > fabs(matrix[imax][ipass]))
            imax = irow;
      }

      /* Exchange ipass row with imax row in both matricies */
      if(imax != ipass) {
         for(icol = 0; icol < size; icol++) {
            SWAP(inverse[ipass][icol], inverse[imax][icol]);
            if(icol >= ipass)
               SWAP(matrix[ipass][icol], matrix[imax][icol]);
         }
      }

      /* Current pivot is now matrix[ipass][ipass] */
      /* Determinant is product of pivot elements */
      /* if det == 0 then matrix is singular and we can't invert */
      pivot = matrix[ipass][ipass];
      det *= pivot;
      if (fabs(det) <= GJ_SING_SENS) {
         /* Singular, so return infinite matrix */
         for (icol = 0; icol < size; icol++)
            for (irow = 0; irow < size; irow++)
               inverse[icol][irow] = FP_PINF;
         return FALSE;
      }

      /* Normalise pivot row by dividing by pivot element */
      for (icol = 0; icol < size; icol++) {
         inverse[ipass][icol] /= pivot;
         if (icol >= ipass)
            matrix[ipass][icol] /= pivot;
      }

      /* Add to each row a mutiple of the pivot row such that */
      /* the element of 'matrix' in the pivot column is zero  */
      for (irow = 0; irow < size; irow++) {
         if (irow != ipass) {
            factor = matrix[irow][ipass];
            for (icol = 0; icol < size; icol++) {
               inverse[irow][icol] -= factor * inverse[ipass][icol];
               matrix[irow][icol] -= factor * matrix[ipass][icol];
            }
         }
      }
   }
   return TRUE;
}

/* --- END of gauss2.c --- */
```

9.3.4 RunAvg

Purpose	Computes a 'running' average, taken over a fixed number of double-precision samples. All samples are initialised to zero. The sampling interval is handled by the user.

Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	December, 1999
Modified	10th October, 2000
Module version	0.02
Other required modules	StdDefs

RunAvg.h

```
/* RunAvg.h -- Computes and uses running averages
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 * Date: December 1999
 * Modified: 10th October, 2000
 * File version: 0.02
 */

#ifndef  __runavg_h
#define  __runavg_h

/* Running average structures */

typedef  struct runAvgTag  runAvg;

/* -- Function InitRunningAverage
 * Pre: 'numSamples' is the number of samples to average over
 * Post: (A new running average variable was created and returned) ||
 *       (There was a problem with allocation && NULL was returned)
 */
runAvg *InitRunningAverage(int numSamples);

/* -- Function AddToRunningAverage
 * Pre: 'avg' has been created by 'InitRunningAverage'
 *      'sample' is a sample to add to the running average
 * Post: 'sample' has been added to 'avg'
 */
void AddToRunningAverage(runAvg *avg, double sample);

/* -- Function ClearRunningAverage
 * Pre: 'avg' has been created by 'InitRunningAverage'
 * Post: all the entries in 'avg' have been cleared to zero
 */
void ClearRunningAverage(runAvg *avg);

/* -- Function GetRunningAverage
 * Pre: 'avg' has been created by 'GetRunningAverage'
 * Post: The average has been computed and returned
 */
double GetRunningAverage(runAvg *avg);

/* -- Function DestroyRunningAverage
 * Pre: 'avg' was created by 'InitRunningAverage'
 * Post: 'avg' was destroyed and deallocated
 */
void DestroyRunningAverage(runAvg **avg);

#endif   /* __runavg_h */

/* --- END of RunAvg.h --- */
```

RunAvg.c

```
/* RunAvg.c -- Computes and maintains running averages
 *
 * See RunAvg.h for details
 */

#include "stddefs.h"
#include "runavg.h"
#include <stdlib.h>

struct runAvgTag {
   int      numSamples;
   double   *AVERAGE;
   int      insertAt;
};

runAvg *InitRunningAverage(int numSamples)
```

```
{
   runAvg   *temp;
   int      index;

   if (!(temp = malloc(sizeof(runAvg))))
      return NULL;

   if (!(temp -> AVERAGE = malloc(numSamples * sizeof(double)))) {
      free(temp);
      return NULL;
   }

   temp -> numSamples = numSamples;
   ClearRunningAverage(temp);

   return temp;
}

void ClearRunningAverage(runAvg *avg)
{
   int   index;

   for (index = 0; index < avg -> numSamples; index++)
      avg -> AVERAGE[index] = 0.0;

   avg -> insertAt = 0;
}

void AddToRunningAverage(runAvg *avg, double sample)
{
   avg -> AVERAGE[avg -> insertAt] = sample;
   avg -> insertAt++;
   if (avg -> insertAt >= avg -> numSamples)
      avg -> insertAt = 0;
}

double GetRunningAverage(runAvg *avg)
{
   double   sum;
   int      index;

   sum = 0;
   for (index = 0; index < avg -> numSamples; index++)
      sum += avg -> AVERAGE[index];

   return (sum / avg -> numSamples);
}

void DestroyRunningAverage(runAvg **avg)
{
   free((*avg) -> AVERAGE);
   free(*avg);
}

/* --- END of RunAvg.c --- */
```

9.3.5 htable

Purpose	Brief hash-table implementation.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	9th February, 1999
Modified	
Module version	0.01
Notes	This is an ADO implementation. Only one per program! Could quite easily be converted into a full ADT… Set the type of the data in the table by changing the **hEntry** typedef in **htable.h**.
Other required modules	StdDefs

htable.h

```
/* htable.h -- Brief hash ADO lookup implementation
```

```
 *
 * Author: Dylan Muir
 * Date: 9th February, 1999
 * Modified:
 * Version: 0.01
 */

#ifndef  __HTABLE_H
#define  __HTABLE_H

#include "stddefs.h"

/* -- Hash system constants -- */

#define  CASE_SENSITIVE    FALSE    /* Case sensitive hash? */
#define  MAX_KEY_SIZE      10       /* Maximum key size */
#define  HASH_EMPTY        "00"     /* Empty hash entry */
#define  HASH_DELETED      "DEL"    /* Deleted hash entry */

/* -- End hash system constants -- */
/* -- Hash system typedefs -- */

typedef  int   hEntry;

/* -- End hash system typedefs -- */
/* -- Hash system functions -- */

/* -- Function hashInitTable
 * Pre: table does not exist
 * Post: table was initialised && returen TRUE
 */
bool  hashInitTable(unsigned hashSize);


/* -- Function hashDestroyTable
 * Pre: table exists
 * Post: table was destroyed && returen TRUE
 */
bool  hashDestroyTable(void);


/* -- Function hashInsert
 * Pre: 'key' is a valid key not in table && table not full
 * Post: ('key' in table && returned TRUE) || (error && returned FALSE)
 */
bool  hashInsert(char *key, hEntry index);


/* -- Function hashValue
 * Pre: 'key' is a valid key in table
 * Post: hash 'key' --> 'value' && returned 'value'
 */
hEntry   hashValue(char *key);


/* -- Function hashDelete
 * Pre: 'key' is a valid key in table && table is not empty
 * Post: ('key' not in table && returned TRUE) || (error && returned FALSE)
 */
bool  hashDelete(char *key);


/* -- Function hashLoadFactor
 * Pre: hash system is initiallised
 * Post: the current load factor is returned as an unsigned integer
 *       percentage from 0 to 100
 */
unsigned hashLoadFactor(void);


/* -- Function hashNumEntries
 * Pre: hash system is initiallised
 * Post: the current number of entries is returned
 */
unsigned hashNumEntries(void);


/* -- Function hashIsIn
 * Pre: table exists
 * Post: ('key' in table && returned TRUE) || ('key' not in table && returned FALSE)
 */
bool  hashIsIn(char *key);
```

```
/* -- Function hashIsFull
 * Pre: hash system in initiallised
 * Post: (hash table is full && returned TRUE) || returned FALSE
 */
bool hashIsFull(void);


/* -- Function hashIsEmpty
 * Pre: hash system in initiallised
 * Post: (hash table is empty && returned TRUE) || returned FALSE
 */
bool hashIsEmpty(void);


/* -- Function hashIsValidKey
 * Pre: hash system in initiallised
 * Post: ('key' is valid && returned TRUE) || returned FALSE
 */
bool hashIsValidKey(char *key);


#endif   /* __HTABLE_H */

/* --- END of htable.h --- */
```

htable.c

```
/* htable.c -- Brief hash lookup implementation
 * SEE htable.h for details
 */

#include <string.h>
#include <stdlib.h>
#include "htable.h"
#include "stddefs.h"

/* -- hash system ADO -- */

typedef struct {
   char  key[MAX_KEY_SIZE];
   hEntry   entry;
} htableEntryTag;

struct htableTag {
   unsigned numEntries;
   unsigned hashSize;
   htableEntryTag *table;
} hash_ADO;



/* -- Non-exported hash functons' defintions -- */

unsigned hash(char *text);
htableEntryTag *hashEntry(unsigned index);


/* -- Hash system functions    -- */
/* -- See htable.h for details -- */

/* -- Function hashInitTable */

bool hashInitTable(unsigned hashSize)
{
   int      index;
   htableEntryTag *record;

   hash_ADO.numEntries = 0;
   hash_ADO.hashSize = hashSize;

   hash_ADO.table = (htableEntryTag *) malloc(hashSize * MAX_KEY_SIZE * sizeof(htableEntryTag));

   if (hash_ADO.table == NULL)
      return FALSE;           /* Couldn't allocate table */

   for (index = 0; index < hashSize; index++) {
      strcpy(hashEntry(index) -> key, HASH_EMPTY);
   }
```

```
   return TRUE;
}


/* -- Function hashDestroyTable -- */

bool hashDestroyTable(void)
{
   free(hash_ADO.table);
   return TRUE;
}


/* -- Function hashInsert */

bool hashInsert(char *key, hEntry entry)
{
   unsigned hashed, insert;
   unsigned hashSize;
   bool     contSearch;
   htableEntryTag *record;

   if (!hashIsValidKey(key))
      return FALSE;           /* Failed precondition */

   if (hashIsIn(key))
       return FALSE;             /* Failed precondition */

   if (hashIsFull())          /* Failed precondition */
      return FALSE;

   insert = hashed = hash(key);
   hashSize = hash_ADO.hashSize;
   contSearch = TRUE;

   while (contSearch) {
      /* Insert if either empty or deleted */
      if (  (strcmp(hashEntry(insert) -> key, HASH_EMPTY) == 0) ||
            (strcmp(hashEntry(insert) -> key, HASH_DELETED) == 0)) {
         record = hashEntry(insert);
         strcpy(record -> key, key);
         record -> entry = entry;
         hash_ADO.numEntries++;
         return TRUE;
      }

      insert += 1;            /* Search in next position */

      if (insert >= hashSize)
         insert = 0;       /* Wrap around table */

      if (insert == hashed) {
         contSearch = FALSE;     /* Searched entire table */
         break;
      }

   }

   return FALSE;               /* Shouldn't reach */
}


/* -- Function hashDelete */

bool hashDelete(char *key)
{
   unsigned hashed, insert;
   unsigned hashSize;
   bool     contSearch;

   if (!hashIsValidKey(key))
      return FALSE;           /* Failed precondition */

   if (!hashIsIn(key))
      return FALSE;           /* Failed precondition */

   if (hashIsEmpty())            /* Failed precondition */
      return FALSE;

   insert = hashed = hash(key);
   hashSize = hash_ADO.hashSize;
   contSearch = TRUE;
```

```
   while (contSearch) {
      /* Delete if matches */
      if (strcmp(hashEntry(insert) -> key, key) == 0) {
         strcpy(hashEntry(insert) -> key, HASH_DELETED);
         hash_ADO.numEntries--;
         return TRUE;
      }

      insert += 1;            /* Search in next position */

      if (insert >= hashSize)
         insert = 0;       /* Wrap around table */

      if (insert == hashed) {
         contSearch = FALSE;     /* Searched entire table */
         break;
      }
   }

   return FALSE;              /* Should never get here */
}


/* -- Function hashLoadFactor */

unsigned hashLoadFactor(void)
{
   if (hashIsEmpty())
      return 0;

   else
      return (unsigned) ((hash_ADO.numEntries * 100) / hash_ADO.hashSize);
}


/* -- Function hashNumEntries */

unsigned hashNumEntries(void)
{
   return hash_ADO.numEntries;
}


/* -- Function hashIsIn */

bool hashIsIn(char *key)
{
   unsigned hashed, insert;
   unsigned hashSize;
   bool     contSearch;

   if (!hashIsValidKey(key))
      return FALSE;           /* Failed precondition */

   insert = hashed = hash(key);        /* Find initial hash */
   hashSize = hash_ADO.hashSize;
   contSearch = TRUE;            /* Do search while contSearch */

   while (contSearch) {
      if (strcmp(hashEntry(insert) -> key, key) == 0) return TRUE;  /* Found key */
      if (strcmp(hashEntry(insert) -> key, HASH_EMPTY) == 0) {   /* never inserted */
         contSearch = FALSE;
         break;
      }

      insert += 1;                  /* Search in next position */

      if (insert >= hashSize)
         insert = 0;                /* Wrap around table */

      if (insert == hashed) {
         contSearch = FALSE;        /* Searched entire table */
         break;
      }
   }

   return FALSE;
}

/* -- Function hashValue */
```

```
hEntry   hashValue(char *key)
{
   unsigned hashed, index;
   unsigned hashSize;
   bool     contSearch, found;

   if (!hashIsValidKey(key))
      return 0;            /* Failed precondition */

   if (!hashIsIn(key))
      return 0;            /* Failed precondition */

   index = hashed = hash(key);        /* Find initial hash */
   hashSize = hash_ADO.hashSize;
   contSearch = TRUE;           /* Do search while contSearch */
   found = FALSE;

   while (contSearch) {
      if (strcmp(hashEntry(index) -> key, key) == 0) {
         contSearch = FALSE;     /* Found key */
         found = TRUE;
         break;
      }
      if (strcmp(hashEntry(index) -> key, HASH_EMPTY) == 0) {    /* never inserted */
         contSearch = FALSE;
         break;
      }

      index += 1;                /* Search in next position */

      if (index >= hashSize)
         index = 0;                 /* Wrap around table */

      if (index == hashed) {
         contSearch = FALSE;        /* Searched entire table */
         break;
      }
   }
   if (!found)
      return 0;
   else
      return hashEntry(index) -> entry;
}


/* -- Function hashIsFull */

bool hashIsFull(void)
{
   if (hash_ADO.numEntries == hash_ADO.hashSize)
      return TRUE;

   else
      return FALSE;
}


/* -- Function hashIsFull */

bool hashIsEmpty(void)
{
   if (hash_ADO.numEntries == 0)
      return TRUE;

   else
      return FALSE;
}


/* -- Function hashIsValidKey */

bool hashIsValidKey(char *key)
{
   if (strlen(key) > MAX_KEY_SIZE)
      return FALSE;

   else
      return TRUE;
}


/* -- Non-exported hash functions -- */
```

```
/* -- Function hash
 * Pre: 'text' is a valid nul-terminated string
 * Post: returned has value of string
 */
unsigned hash(char *text)
{
   unsigned hash;

   hash = 0;
   while (*text != EOS) {
      hash += *text;
      text++;
   }

   hash %= hash_ADO.hashSize;

   return hash;
}

/* -- Function hashEntry
 * Pre: 'index' is a valid hash
 * Post: (a pointer to the indexed entry was returned) ||
 *       (an error occurred && NULL was returned)
 */
htableEntryTag *hashEntry(unsigned index)
{
   if (index >= hash_ADO.hashSize)
      return NULL;

   return (hash_ADO.table + index * MAX_KEY_SIZE);
}


/* --- END of htable.c --- */
```

9.3.6 smdarray

Purpose	Enables the use of infinitely-dimensioned arrays in C. When collecting *n*-gram transition data for a (say) 29-class problem, you've got a **big** transition array. You won't be able to allocate it. This module manages sparse any-dimensioned arrays.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	22nd September, 1999
Modified	6th October, 1999
Module version	0.04
Notes	In this module, only integers can be stored in the array. This is relatively easy to modify in the code, but no **typedef** statement exists to do it. Consider this a limitation of the module at present.
Other required modules	StdDefs

Examples

Create the array by using **sArrayCreate**. The example we will use is an array **saTransition**[29][29][29][29].

```
smdarray  saTransition;
saTransition = sArrayCreate();
```

This will return **NULL** if it fails to allocate the array. To access the array, you must specify the number of dimensions it has in the tool function call.

```
sArray...(saTranstion, 4, ..., i, j, k, m);
```

The four dimensions are **i**, **j**, **k** and **m**. To use a higher number of dimensions, insert the number of dimensions in place of '4' and append the extra dimensions to the end of the function call. Like **printf**, the **sArray...** functions use variable-length argument lists.

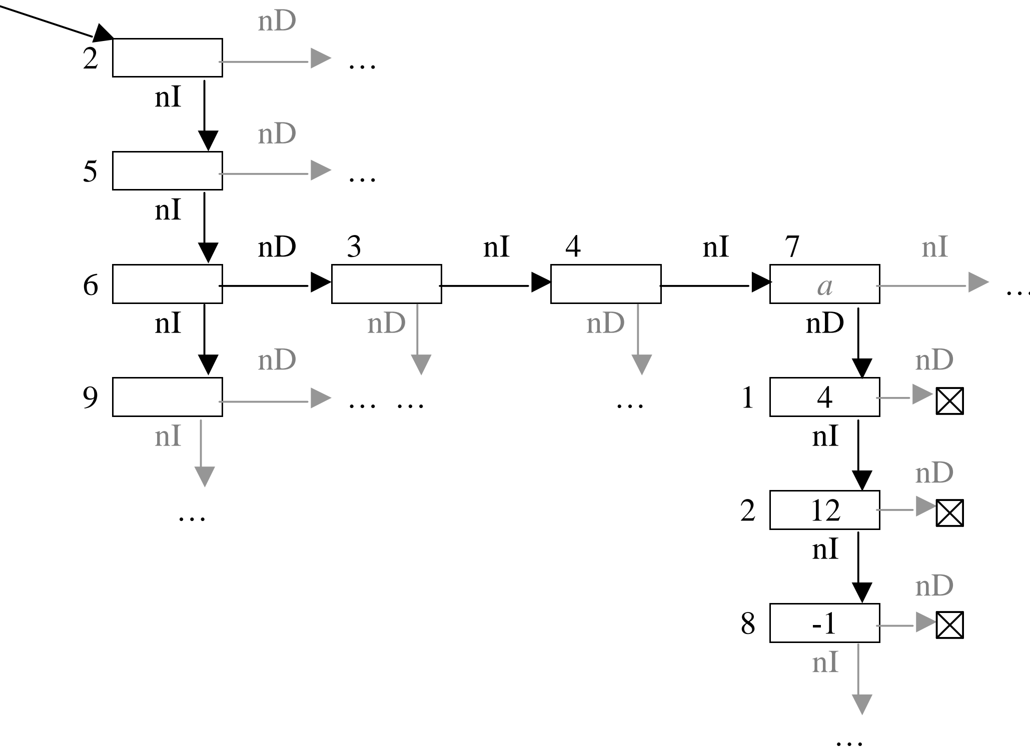
To insert something in our array, we'll use **sArrayAssign**.

```
sArrayAssign(saTransition, 4, -1, 8, 23, 12, 81);
```

This call assigns the value '-1' to **saTransition**[8][23][12][81]. It will create the cell in the array if it doesn't already exist. At this stage, only the cell at **saTransition**[8][23][12][81] exists. The function will return **FALSE** if it fails to allocate a cell.

Data structure explanation

Each node in the structure corresponds to an index along a dimension in the array. Each node has an index associated with it, a place to insert a value if the node is a leaf, a link along to the next index in the current dimension and a link to the next dimension if the node is not a leaf. The number of dimensions that have been used to create the array is not stored, and the responsibility to ensure that the indices are valid falls to the user.



In the diagram above, an array with three dimensions is being accessed. The target cell is **smArray**[6][7][8]. The first dimension is traversed from the root of the structure. When the desired index is reached, the **nextDimension** index is traversed to obtain the list corresponding to the higher dimension. This list is then traversed until the second index (7) is reached. The higher dimension is again accessed, and the list traversed until the leaf corresponding to index (8) is reached. The data stored in this cell is accessed.

Note that there is no data stored in cells that do not represent the final dimension. As mentioned above, this is **not** enforced by the module. It is up to the user to ensure that the dimensionality of arrays is consistent. If the array was accessed using two dimensions, for example **smArray**[6][7], then the cell *a* would be accessed.

smdarray.h

```
/* smdarray -- Sparse Multi-Dimensional Arrays
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 22nd September, 1999
 * Modified: 27th September, 1999 (hbtm)
 *           6th October, 1999
 * Version: 0.04
 *
 * Description: This module allows for the use of infinitely-dimensioned
 *              sparse arrays.  You'll have problems in C when you start
 *              doing 4- and 5-gram transition tables of 29-class data.
 *              Allocating the array is HUGE (29 * 29 * 29 * 29 = 707281
 *              cells for 4-gram) and most of the elements won't even be
 *              used!  This module provides the solution.
 */

#ifndef  __smdarray_h
#define  __smdarray_h

/* -- smdarray defines -- */

#define  smd_MAX_DIMENSIONS   100

/* -- smdarray required modules -- */

#include "stddefs.h"


/* -- smdarray structure -- */

typedef struct smdArrayTag {
   int   index,            /* index in current dimension */
         cell;             /* contents of current cell */

   struct smdArrayTag   *nextCell,        /* iterator within dimension */
                        *nextDimension;   /* pointer to higher dimension */
} *smdarray;               /* default typename to use */


/* -- smdArray functions -- */

/* -- Function sArrayCreate
 * Pre: TRUE
 * Post: returns an empty smdarray or NULL if creation failed
 */
smdarray sArrayCreate(void);

/* -- Function sArrayDestroy
 * Pre: 'array' is a valid smdarray
 * Post: all cells in 'array' were destroyed && an empty array was returned
 */
smdarray sArrayDestroy(smdarray array);

/* -- Function sArrayAccess
 * Pre: 'array' is  avalid created smd array
 *      'numDimensions' is the total number of dimensions you are accessing
 *      '...' is a list of indices into 'array'
 * Post: (a value has been assigned to 'array[...]' && the value was returned) ||
 *       (a value has not been assigned to 'array[...]' && zero was returned)
 */
int sArrayAccess(smdarray array, int numDimensions, ...);

/* -- Function sArrayGetCell
 * Pre: 'array' is a valid created smd array
 *      'numDimensions' is the number of dimensions to access in 'array'
 *      '...' is a list of indices into 'array'
 * Post: (a cell corresponding to 'array[...]' was returned) ||
 *       (there was a problem with allocation && NULL was returned)
 * Note: Since sArrayGetCell will create a cell if one doesn't exist, indexing
 *       through 'array' using sArrayGetCell would be exceedingly stupid, since
 *       when you finished you wouldn't have a sparse array anymore.
 *       You probably shouldn't index along .nextCell and .nextDimension.
 */
smdarray sArrayGetCell(smdarray array, int numDimensions, ...);

/* -- Funcion sArrayDoesCellExist
 * Pre: 'array' is a valid created smd array
 *      'numDimensions' is the number of dimensions to access in 'array'
 *      '...' is a list of indices into 'array'
```

```
 * Post: ('array[...]' has had a value assigned to it && TRUE was returned) ||
 *       ('array[...]' has never had a value assigned to it && FALSE was returned))
 */
bool sArrayDoesCellExist(smdarray array, int numDimensions, ...);

/* -- Function sArrayAssign
 * Pre: 'array' is a valid created smd array
 *      'numDimensions' is the number of dimensions to access in 'array'
 *      'value' is the value to assign to 'array[...]'
 *      '...' is a list of indices into 'array'
 * Post: ('value' was assigned to 'array[...]' && TRUE was returned) ||
 *       (there was a problem with allocation && FALSE was returned)
 */
bool sArrayAssign(smdarray array, int numDimensions, int value, ...);

/* -- Function sArrayIncrement
 * Pre: 'array' is a valid created smd array
 *      'numDimensions' is the number of dimensions to access in 'array'
 *      'delta' is a number to add arithmetically to 'array[...]'
 *      '...' is a list of indices into 'array'
 * Post: ('array[...]' was incremented by 'delta' && TRUE was returned) ||
 *       (there was a problem with allocation && FALSE was returned)
 */
bool sArrayIncrement(smdarray array, int numDimensions, int delta, ...);

/* -- Function sArrayCountLeaves
 * Pre: 'cell' is a valid cell in an smd array
 * Post: the number of leaves in subdimensions off 'cell' was returned
 */
int sArrayCountLeaves(smdarray cell);

/* -- Function sArrayBranchWeight
 * Pre: 'cell' is a valid cell in an smd array
 * Post: the values of all the leaves of 'cell' were summed and returned
 */
int sArrayBranchWeight(smdarray cell);

#endif   /* __smdarray_h */

/* --- END of smdarray.h --- */
```

smdarray.c

```
/* smdarray -- Sparse, multi-dimensional arrays
 *
 * see smdarray.h for details
 */

/* -- smdarray includes -- */

#include "smdarray.h"
#include "stddefs.h"
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>


/* -- smdarray helper functions prototypes -- */

/* -- Function sArrayEnsureCellExistsDiscrete
 * Pre: 'array' is a valid created smd array
 *      'numDimensions' is the number of dimensions to access in 'array'
 *      'indices' is an integer array of the indices, of order 'numDimensions'
 * Post: (a cell representing 'array[indices]' was returned) ||
 *       (there was a problem with allocation && NULL was returned)
 */
smdarray sArrayEnsureCellExistsDiscrete(smdarray array, int numDimensions, int indices[]);

/* -- Function sArrayDoesCellExistDiscrete
 * Pre: 'array' is a valid created smd array
 *      'numDimensions' is the number of dimensions to access in 'array'
 *      'indices' is an integer array of the indices, of order 'numDimensions'
 * Post: ('array[indices]' has had a alue assigned to it && TRUE was returned) ||
 *       ('array[indices]' has not had a value assigned to it && FALSE was returned)
 */
bool sArrayDoesCellExistDiscrete(smdarray array, int numDimensions, int indices[]);

/* -- Function sArrayQueryCellDiscrete
 * Pre: 'array' is a valid created smd array
 *      'numDimensions' is the number of dimensions to access in 'array'
 *      'indices' is an integer array of the indices, of order 'numDimensions'
```

```
 * Post: (a cell representing 'array[indices]' was returned if one exists) ||
 *       (the cell doesn't exists && NULL was returned)
 */
smdarray sArrayQueryCellDiscrete(smdarray array, int numDimensions, int indices[]);


/* -- Function sArrayMakeCell
 * Pre: 'index' and 'value' are reasonable integers
 * Post: (a cell was created, 'index' and 'value' were assigned to it, the
 *       nextCell and nextDimension indices were set to NULL && the cell was returned) ||
 *       (there was a problem with allocation && NULL was returned)
 */
smdarray sArrayMakeCell(int index, int value);


/* -- smdarray functions -- */

smdarray sArrayCreate(void)
{
   smdarray array;

   return sArrayMakeCell(0, 0);
}

int sArrayAccess(smdarray array, int numDimensions, ...)
{
   va_list  pindex;
   int   *indices,
      dIndex;

   if (!(indices = (int *) malloc(sizeof(int) * numDimensions)))
      return 0;

   va_start(pindex, numDimensions);
   dIndex = 0;
   while (dIndex < numDimensions) {
      indices[dIndex] = va_arg(pindex, int);
      dIndex++;
   }
   va_end(pindex);

   if (!sArrayDoesCellExistDiscrete(array, numDimensions, indices))
      return 0;

   return sArrayQueryCellDiscrete(array, numDimensions, indices) -> cell;
}

smdarray sArrayGetCell(smdarray array, int numDimensions, ...)
{
   va_list  pindex;
   int   *indices,
      dIndex;

   if (!(indices = (int *) malloc(sizeof(int) * numDimensions)))
      return NULL;

        va_start(pindex, numDimensions);
        dIndex = 0;
        while (dIndex < numDimensions) {
                indices[dIndex] = va_arg(pindex, int);
                dIndex++;
        }
        va_end(pindex);

   return sArrayEnsureCellExistsDiscrete(array, numDimensions, indices);
}

bool sArrayDoesCellExist(smdarray array, int numDimensions, ...)
{
        va_list pindex;
        int     *indices,
                dIndex;

        if (!(indices = (int *) malloc(sizeof(int) * numDimensions)))
                return FALSE;

        va_start(pindex, numDimensions);
        dIndex = 0;
        while (dIndex < numDimensions) {
                indices[dIndex] = va_arg(pindex, int);
                dIndex++;
        }
```

```
        va_end(pindex);

   return sArrayDoesCellExistDiscrete(array, numDimensions, indices);
}

bool sArrayAssign(smdarray array, int numDimensions, int value, ...)
{
        va_list   pindex;
        int       *indices,
                  dIndex;
   smdarray cell;

        if (!(indices = (int *) malloc(sizeof(int) * numDimensions)))
                return FALSE;

        va_start(pindex, value);
        dIndex = 0;
        while (dIndex < numDimensions) {
                indices[dIndex] = va_arg(pindex, int);
                dIndex++;
        }
        va_end(pindex);

   if (!(cell = sArrayEnsureCellExistsDiscrete(array, numDimensions, indices)))
      return FALSE;

   cell -> cell = value;
   return TRUE;
}

smdarray sArrayDestroy(smdarray array)
{
   if (array -> nextDimension != NULL)
      array -> nextDimension = sArrayDestroy(array -> nextDimension);
   if (array -> nextCell != NULL)
      array -> nextCell = sArrayDestroy(array -> nextCell);

   free(array);
   return NULL;
}

bool sArrayIncrement(smdarray array, int numDimensions, int delta, ...)
{
        va_list   pindex;
        int       *indices,
                  dIndex;
   smdarray cell;

        if (!(indices = (int *) malloc(sizeof(int) * numDimensions)))
                return FALSE;

        va_start(pindex, delta);
        dIndex = 0;
        while (dIndex < numDimensions) {
                indices[dIndex] = va_arg(pindex, int);
                dIndex++;
        }
        va_end(pindex);

   if (!(cell = sArrayEnsureCellExistsDiscrete(array, numDimensions, indices)))
      return FALSE;

   cell -> cell += delta;
   return TRUE;
}

int sArrayCountLeaves(smdarray cell)
{
   int   count;

   count = 0;

   if (cell -> nextDimension == NULL)
      count += 1;
   else
      count += sArrayCountLeaves(cell -> nextDimension);

   if (cell -> nextCell != NULL)
      count += sArrayCountLeaves(cell -> nextCell);

   return count;
}
```

```
int sArrayBranchWeight(smdarray cell)
{
   int   weight;

   weight = 0;

   if (cell -> nextDimension == NULL)
      weight += cell -> cell;
   else
      weight += sArrayBranchWeight(cell -> nextDimension);

   if (cell -> nextCell != NULL)
      weight += sArrayBranchWeight(cell -> nextCell);

   return weight;
}

/* -- smdarray helper functions -- */

bool sArrayDoesCellExistDiscrete(smdarray array, int numDimensions, int indices[])
{
   return (sArrayQueryCellDiscrete(array, numDimensions, indices) != NULL);
}

smdarray sArrayQueryCellDiscrete(smdarray array, int numDimensions, int indices[])
{
   int      dIndex;
   smdarray cell;

   cell = array;
   dIndex = 0;
   while (dIndex < numDimensions) {
      while (cell -> index < indices[dIndex]) {
         if (cell -> nextCell == NULL)
            return NULL;
         cell = cell -> nextCell;
      }
      if (cell -> index != indices[dIndex])
         return NULL;
      dIndex++;
      if (dIndex < numDimensions) {
         if (cell -> nextDimension == NULL)
            return NULL;
         cell = cell -> nextDimension;
      }
   }
   return cell;
}


smdarray sArrayEnsureCellExistsDiscrete(smdarray array, int numDimensions, int indices[])
{
   int      dIndex;
   smdarray cell, newCell;

   if (sArrayDoesCellExistDiscrete(array, numDimensions, indices))
      return sArrayQueryCellDiscrete(array, numDimensions, indices);

   cell = array;
   dIndex = 0;
   while (dIndex < numDimensions) {
      while (cell -> index < indices[dIndex]) {
         if ((cell -> nextCell == NULL) || (cell -> nextCell -> index > indices[dIndex])) {
            if (!(newCell = sArrayMakeCell(indices[dIndex], 0)))
               return NULL;
            newCell -> nextCell = cell -> nextCell;
            cell -> nextCell = newCell;
         }
         cell = cell -> nextCell;
      }
      dIndex++;
      if (dIndex < numDimensions) {
         if ((cell -> nextDimension == NULL) || (cell -> nextDimension -> index > indices[dIndex]))
{
            if (!(newCell = sArrayMakeCell(indices[dIndex], 0)))
               return NULL;
            newCell -> nextCell = cell -> nextDimension;
            cell -> nextDimension = newCell;
         }
         cell = cell -> nextDimension;
      }
```

```
   }

   if (!sArrayDoesCellExistDiscrete(array, numDimensions, indices)) {
      printf("houston, we have a problem.\n");
      return NULL;
   }
   return sArrayQueryCellDiscrete(array, numDimensions, indices);
}

smdarray sArrayMakeCell(int index, int value)
{
   smdarray newCell;

   if (!(newCell = (smdarray) malloc(sizeof(struct smdArrayTag))))
      return NULL;

   newCell -> index = index;
   newCell -> cell = value;
   newCell -> nextCell = NULL;
   newCell -> nextDimension = NULL;

   return newCell;
}


/* --- END of smdarray.c --- */
```

9.3.7 TokenLst

Purpose	Separates whitespace-separated words into separate tokens.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	29th September, 1998
Modified	14th September, 1999
Module version	1.11
Notes	Could be expanded to include different separators. Need to modify TokenLst.c for this.
Other required modules	StdDefs

TokenLst.h

```
/* TokenLst.h -- module containing Token List ADT and token scanner
 *
 * Author: Dylan Muir
 * Created: 29th September, 1998
 * Modified: 14th September, 1999
 * Version: 1.11
 *
 * Description: A tokenlist is an ordered list of tokens.  A token list is maintained in the
 *              same order as the tokens were added.  Tokens are separated by whitespace.
 */

#ifndef  __TOKENLST_H
#define  __TOKENLST_H

/* -- TokenList data structure -- */

struct tokenListTag {
   char  *token;        /* Token text */

   struct tokenListTag  *NEXT;      /* Next token */
};

typedef struct tokenListTag   tokenList;


/* -- TokenList support functions -- */

/* -- Function GetTokens
 * Pre: 'text' is a valid string
 * Post: (All tokens were scanned from 'text' and returned as a list) ||
 *       (the operation was unsuccesful && NULL was returned)
 */
tokenList *GetTokens(char *text);
```

```
/* -- Function AddToken
 * Pre: 'tokenText' is a valid string && 'list' is a valid TokenList
 * Post: A token containing 'tokenText' was allocated and added to the end of 'list'
 */
tokenList *AddToken(char tokenText[], tokenList *list);

/* -- Function DestroyTokenList
 * Pre: 'list' is a valid TokenList
 * Post: 'list' was deallocated && an empty TokenList was returned
 */
tokenList *DestroyTokenList(tokenList *list);


/* -- Function ConcatenateTokenList
 * Pre: 'list1' and 'list2' are valid TokenLists
 * Post: 'list2' was appended to 'list1' and the resulting list was returned
 */
tokenList *ConcatenateTokenList(tokenList *list1, tokenList *list2);

/* -- Function TokenListLength
 * Pre: 'list' is a valid NULL-terminated token list
 * Post: the length of 'list' has been counted and was returned
 */
int TokenListLength(tokenList *list);

#endif   /* __TOKENLST_H */
```

TokenLst.c

```
/* TokenLst.c -- Module containing Token list ADT and token scanner
 *
 * See include file for details
 */
#include "tokenlst.h"
#include "stddefs.h"

#include <string.h>
#include <malloc.h>

#define  CreateString(length) ((char *) malloc(length))

/* -- Non publicly declared function prototypes -- */

/* char  *CreateString(int length); */
tokenList *CreateToken(void);


/* -- Publicly Declared functions -- */

/* -- Function GetTokens
 */
tokenList *GetTokens(char *text)
{
   bool  inToken;                /* Are we currently in a token? */
   char  read,                   /* Last read character */
         *tokenBuffer,           /* Buffer for reading a token */
         *index;                 /* index into tokenBuffer */

   tokenList   *list;            /* Token list to return */

   if ((tokenBuffer = CreateString(MAXSTRING)) == NULL) /* Allocate token buffer */
      return NULL;

   list = NULL;                     /* Reset token list */
   index = tokenBuffer;             /* Reset index */

   inToken = FALSE;     /* Reset flags */

   while (TRUE) {                /* All returns are from within loop */
      read = *text++;

      switch (read) {
         case EOS:                     /* End of string */
            if (inToken) {
               *index = EOS;                 /* Nul-terminate */
               inToken = FALSE;              /* No longer in a token */
               index = tokenBuffer;          /* Reset index */

               list = AddToken(tokenBuffer, list); /* Add token to list */
            }
```

```
            free (tokenBuffer);          /* Deallocate token buffer */

            return list;                 /* Exit, return token list */
            /* break; */

         case ' ':                    /* Whitespace */
         case TAB:
            if (inToken) {
               inToken = FALSE;              /* No longer in a token */
               *index = EOS;                 /* Nul-terminate string */
               index = tokenBuffer;          /* Reset index */

               list = AddToken(tokenBuffer, list); /* Add token to list */
            } else
               ;                             /* not in a token, so ignore */
            break;

         default:
            if (!inToken)             /* If we're not in a token, then... */
               inToken = TRUE;           /* start of a new token */

            *index++ = read;        /* In a token, so copy */

            break;
      }
   }
}


/* -- Function AddToken
 */
tokenList *AddToken(char tokenText[], tokenList *list)
{
   tokenList   *insert,                  /* index to insert option at */
               *index;

   if ((insert = CreateToken()) == NULL)  /* Can we create a token? */
      return NULL;

   strncpy(insert -> token, tokenText, MAXSTRING - 1);   /* Copy token text */

   index = list;                         /* Reset index into token list */

   if (index != NULL) {                  /* Is it a null list */
      while (index -> NEXT != NULL)         /* Loop until end of list */
         index = index -> NEXT;                /* Step through list */

      index -> NEXT = insert;               /* Insert token at end of list */
      return list;                           /* return list */
   }

   return insert;                        /* Otherwise just return token (only node in list) */
}


/* -- Function DestroyTokenList
 */
tokenList *DestroyTokenList(tokenList *list)
{
   if (list != NULL) {                   /* Is it not a null list? */
      DestroyTokenList(list -> NEXT);       /* Recursive call */
      free (list -> token);                 /* Free token text */
      free (list);                          /* Free token structure */
   }
   return NULL;                          /* return an empty list */
}


tokenList *ConcatenateTokenList(tokenList *list1, tokenList *list2)
{
   tokenList   *index;

   if (list2 == NULL)         /* No list 2 */
      return list1;
   if (list1 == NULL)         /* No list 1 */
      return list2;

   index = list1;

   while (index -> NEXT != NULL)    /* Shuffle to end of list */
      index = index -> NEXT;
```

```
   index -> NEXT = list2;

   return list1;           /* Return the resulting list */
}


int TokenListLength(tokenList *list)
{
   int   length = 0;

   while (list != NULL) {
      length++;
      list = list -> NEXT;
   }
   return length;
}

/* -- End of publicly declared functions -- */

/* --- Function CreateString
 * Pre: TRUE
 * Post: (Memory for a string of length 'length' was allocate dand returned) ||
 *       (There was insufficient memory && NULL was returned)
 *
char  *CreateString(int length)
{
   return (char *) malloc(length);
}*/


/* -- Function CreateToken
 * Pre: TRUE
 * Post: (A token was allocated and returned) ||
 *       (The operation was unsuccessful && NULL was returned)
*/
tokenList *CreateToken(void)
{
   tokenList   *tempToken;    /* Temporary token for allocation */

   if ((tempToken = (tokenList *) malloc(sizeof(tokenList))) == NULL)
      return NULL;

   if ((tempToken -> token = CreateString(MAXSTRING)) == NULL) {
      free (tempToken);
      return NULL;
   }

   tempToken -> NEXT = NULL;

   return tempToken;          /* Send back token */
}

/* -- END of TokenLst.C -- */
```

9.3.8 TokScan

Purpose	Uses the token list data structure to retrieve tokens from a file.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	29th September, 1998
Modified	10th May, 1999
Module version	1.01
Other required modules	StdDefs, TokenLst

TokScan.h

```
/* TokScan.h -- Token scanning engine
 *
 * Author: Dylan Muir [dr.muir@student.qut.edu.au]
 * Date: 29th September, 1998
 * Modified: 10th May, 1999
 * Version: 1.01
 */

#ifndef  __TOKSCAN_H
#define  __TOKSCAN_H
```

```c
#include <stdio.h>
#include "tokenlst.h"
#include "stddefs.h"


/* -- Function ReadLineTokens
 * Pre: 'file' is a text file opened for reading
 *      'inFile' has been allocated, and indicates wether EOF has been reached
 * Post: a line of text up to an EOL or EOF has been scanned and
 *       converted into a token list, which was returned
 */
tokenList   *ReadLineTokens(FILE *ifile, bool *inFile);

/* -- Function ReadLineText
 * Pre: 'file' is a text file opened for reading
 *      'buffer' has been allocated, and has max length 'length'
 *      'inFile' has been allocated, and indicated wether EOF has been reached
 * Post: 'buffer' contains a line of text up to EOL or EOF (defined in stddefs.h) &&
 *       'inFile' indicates whether EOF has been reached
 */
void  ReadLineText(FILE *ifile, char *buffer, unsigned length, bool *inFile);

#endif   /* __TOKSCAN_H */

/* --- END of TokScan.h --- */
```

TokScan.c

```c
/* tokscan.c -- Token scanning engine
 *
 * See tokscan.h for details
 */

#include "stddefs.h"
#include "tokscan.h"
#include "tokenlst.h"

/* -- Exported functions -- */

tokenList *ReadLineTokens(FILE *ifile, bool *inFile)
{
   char  buffer[MAXSTRING * 4];

   ReadLineText(ifile, buffer, MAXSTRING * 4, inFile);

   return GetTokens(buffer);
}

void ReadLineText(FILE *ifile, char *buffer, unsigned length, bool *inFile)
{
   char        read;                    /* Character read from 'file' */
   unsigned    count;                   /* number of characters in 'buffer' */

   count = 0;

   fread(&read, 1, 1, ifile);

   while (read != EOL && *inFile) {
      if (feof(ifile))
         *inFile = FALSE;

      else {
         *buffer++ = read;
         count++;

         if (count == length - 1)      /* Filled the buffer */
            break;
      }

      fread(&read, 1, 1, ifile);
   }

   *buffer = EOS;
}


/* --- END of TokScan.c --- */
```

9.3.9 vector_utils

Purpose	A series of functions for dealing with vectors.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	14th September, 1999
Modified	13th July, 2000
Module version	0.12
Other required modules	2darray

vector_utils.h

```
/* Vector utils: a series of functions for dealing with vectors
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 14th September, 1999
 * Modified: 13th July, 2000
 * Version: 0.12
 */

#ifndef __vector_utils_h
#define  __vector_utils_h

/* -- Vector includes */

#include "stddefs.h"

/* -- Vector structures */

typedef struct {
   int      dimensions;
   double   *components;
} vector;


/* -- Vector functions */

/* -- Function VectorAllocate
 * Pre: 'dimensions' is the number of dimensions to allocate for this vector
 * Post: (The vector was allocated && returned) ||
 *       (there was an error allocating the vector && NULL was returned)
 * Note: a new vector has undefined components
 */
vector *VectorAllocate(int dimensions);

/* -- Function VectorConstruct
 * Pre: 'dimensions' is the number of dimensions to allocate for this vector
 *      'components' is a double array of components to place in this vector
 * Post: (A new vector was allocated based on 'components', the contents of
 *        'components' was copied into the vector && the vector was returned) ||
 *       (There was an error allocating the vector && NULL was returned)
 */
vector *VectorConstruct(int dimensions, const double *components);

/* -- Function VectorDeallocate
 * Pre: 'vect' is a valid allocated vector
 * Post: 'vect' was deallocated && NULL was returned
 */
vector *VectorDeallocate(vector *vect);

/* -- Function VectorEuclideanDist
 * Pre: 'vect1' and 'vect2' are valid allocated vectors
 *      the dimensionality of 'vect1' and 'vect2' is the same
 */
double VectorEuclideanDist(vector *vect1, vector *vect2);

/* -- Function VectorLength
 * Pre: 'vect' is a valid allocated vector
 * Post: returned the euclidean distance from 0 to 'vect'
 */
double VectorLength(vector *vect);

/* -- Function VectorZero
 * Pre: 'dimensions' > 0
 * Post: (returned a reference to a zero vector) ||
 *       (there was an error in allocation && returned NULL)
```

```
 * Note: VectorZero allocates a new vector, which you MUST deallocate!
 */
vector *VectorZero(int dimensions);

/* -- Function VectorUnit
 * Pre: 'dimensions' > 0
 * Post: (returned a reference to a unit vector in 'dimensions' dimensions) ||
 *       (there was an error in allocation && returned NULL)
 * Note: VectorZero allocates a new vector, which you MUST deallocate!
 */
vector *VectorUnit(int dimensions);

/* -- Function VectorDotProduct
 * Pre: 'vect1' and 'vect2' are valid allocated vectors
 *      the dimensionality of 'vect1' and 'vect2' is equal
 * Post: returned the dot product of 'vect1' and 'vect2'
 * Note: The name for this function is a little ambiguous;
 *       all the functions in the vector suite are prepended
 *       with 'Vector', but this function performs the
 *       _scalar_ dot product of two vectors.  There's no
 *       thing as a vector dot product, is there?
 */
double VectorDotProduct(vector *vect1, vector *vect2);

/* -- Function VectorSum
 * Pre: 'vectd' and 'vects' are valid allocated vectors
 *      the dimensionality of 'vectd' and 'vects' is the same
 * Post: 'vectd' = 'vectd' + 'vects'
 */
void VectorSum(vector *vectd, vector *vects);

/* -- Function VectorSub
 * Pre: 'vectd' and 'vects' are valid allocated vectors
 *      the dimensionality of 'vectd' and 'vects' is the same
 * Post: 'vectd' = 'vectd' - 'vects'
 */
void VectorSub(vector *vectd, vector *vects);

/* -- Function VectorNeg
 * Pre: 'vect' is a valid allocated vector
 * Post: 'vect' = -'vect'
 */
void VectorNeg(vector *vect);

/* -- Function VectorSquare
 * Pre: 'vect' is a valid allocated vector
 * Post: arg(i) 'vect'i = 'vect'i * 'vect'i
 *       i.e. each element of the vector is squared
 * Note: Performs x.^2
 */
void VectorSquare(vector *vect);

/* -- Function VectorMultiplyScalar
 * Pre: 'vect' is a valid allocated vector
 *      'scalar' is a valid number
 * Post: 'vect' = 'vect' * 'scalar'
 */
void VectorMultiplyScalar(vector *vect, double scalar);

/* -- Function VectorDivideScalar
 * Pre: 'vect' is a valid allocated vector
 *      'scalar' is a valid number
 * Post: 'vect' = 'vect' / 'scalar'
 */
void VectorDivideScalar(vector *vect, double scalar);

/* -- Function VectorMakeCopy
 * Pre: 'vect' is a valid allocated vector
 * Post: (a copy of 'vect' was allocated and returned ) ||
 *       ( there was an error with allocation && NULL was returned)
 */
vector *VectorMakeCopy(vector *vect);

/* -- Function VectorMakeHomogeneous
 * Pre: 'vect' is a valid allocated vector
 * Post: (a new vector was created containing ['vect', 1.0]) ||
 *       (there was an error with allocation && NULL was returned)
 */
vector *VectorMakeHomogeneous(vector *vect);

/* -- Function VectorMaxComponent
 * Pre: 'vect' is a valid allocated vector
```

```
 * Post: the 0-offset index of the maximum component of 'vect' was returned
 * Note: in the case of a tie for maximum, one of the indicies is returned
 */
int VectorMaxComponent(vector *vect);

/* -- Function VectorIdentical
 * Pre: 'vect1' and 'vect2' are valid allocated vectors
 * Post: (all components of 'vect1' are the same as 'vect2' && TRUE was returned) ||
 *       (FALSE was returned)
 */
bool VectorIdentical(vector *vect1, vector *vect2);

/* -- Function VectorManhattanEntropy
 * Pre: 'vect1' and 'vect2' are valid allocated vectors
 *      dim('vect1') == dim('vect2')
 * Post: The Manhattan Distance Entropy for 'vect1' and 'vect2' was
 *       calculated and returned
 * Note: H = Sum over dimensions(fd ln(fd))
 *       Where fd = |ad - bd| / Sum over d(|ad - bd|)
 */
double VectorManhattanEntropy(vector *vect1, vector *vect2);

/* -- Function VectorCorrelation
 * Pre: 'vect' is a valid allocated vector
 *      'matrixSize' points to an allocated integer
 * Post: The correlation matrix of 'vect' is returned as a 2D array of double.
 *       The matrix is square; the size of the matrix is placed in 'matrixSize'.
 *       NULL is returned if allocation is not possible.  In this case,
 *       'matrixSize' is undefined.
 *
 * Note: 'vect' = [a b c]
 *
 *           R = [a][a b c]
 *               [b]
 *               [c]
 *
 *             = [a2 ab ac]
 *               [ab b2 bc]
 *               [ac bc c2]
 */
double **VectorCorrelation(vector *vect, int *matrixSize);


#endif /* __vector_utils_h */


/* --- END of vector_utils.h --- */
```

vector_utils.c

```
/* Vector utils: a series of functions for dealing with vectors
 *
 * SEE vector_utils.h for details
 */
#include "vector_utils.h"
#include "2darray.h"
#include <stdlib.h>
#include <math.h>
#include <memory.h>
#ifdef   UNIX
   #include <ieeefp.h>
#endif
#ifdef   WIN32
   #include <float.h>
   #define  FP_SNAN  _FPCLASS_SNAN
#endif

#include <stdio.h>

/* -- Vector macros -- */

#define  SQR(x)   ((x) * (x))
#define  ABS(x)   ((x) < 0 ? (-(x)) : (x))


/* -- Vector functions -- */

vector *VectorAllocate(int dimensions)
{
   vector   *vect;
```

```
   if ((vect = (vector *) malloc(sizeof(vector))) == NULL)
      return NULL;

   if ((vect -> components = (double *) malloc(dimensions * sizeof(double))) == NULL) {
      free(vect);
      return NULL;
   }

   vect -> dimensions = dimensions;
   return vect;
}

vector *VectorConstruct(int dimensions, const double *components)
{
   vector   *newVect;

   if (!(newVect = VectorAllocate(dimensions)))
      return NULL;

   memcpy(newVect -> components, components, dimensions * sizeof(double));
   return newVect;
}


vector *VectorDeallocate(vector *vect)
{
   free(vect -> components);
   free(vect);
   return NULL;
}

double VectorLength(vector *vect)
{
   vector   *zero;
   double   length;

   if ((zero = VectorZero(vect -> dimensions)) == NULL)
      return 0.0;

   length = VectorEuclideanDist(vect, zero);
   VectorDeallocate(zero);
   return length;
}

double VectorEuclideanDist(vector *vect1, vector *vect2)
{
   double   dimensionSum;
   int   dimension;

   if (vect1 -> dimensions != vect2 -> dimensions)
      return 0.0;

   dimensionSum = 0.0;
   dimension = 0;
   while (dimension < vect1 -> dimensions) {
      dimensionSum += SQR(vect1 -> components[dimension] - vect2 -> components[dimension]);
      dimension++;
   }

   return sqrt(dimensionSum);
}

vector *VectorZero(int dimensions)
{
   vector   *zero;
   int   compIndex;     /* index into components */

   if ((zero = VectorAllocate(dimensions)) == NULL)
      return NULL;

   compIndex = 0;
   while (compIndex < dimensions) {
      zero -> components[compIndex] = 0.0;
      compIndex++;
   }
   return zero;
}

vector *VectorUnit(int dimensions)
{
   vector   *unit;
   int   compIndex;     /* index into componenets */
```

```
   if ((unit = VectorAllocate(dimensions)) == NULL)
      return NULL;

   compIndex = 0;
   while (compIndex < dimensions) {
      unit -> components[compIndex] = 1 / (double) dimensions;
      compIndex++;
   }
   return unit;
}

double VectorDotProduct(vector *vect1, vector *vect2)
{
   double   productSum;
   int   dimension;

   if (vect1 -> dimensions != vect2 -> dimensions)
      return 0.0;

   productSum = 0.0;
   dimension = 0;
   while (dimension < vect1 -> dimensions) {
      productSum += vect1 -> components[dimension] * vect2 -> components[dimension];
      dimension++;
   }

   return productSum;
}

void VectorSum(vector *vectd, vector *vects)
{
   int   dIndex;

   if (vectd -> dimensions != vects -> dimensions)
      return;

   dIndex = vectd -> dimensions - 1;
   while (dIndex >= 0) {
      vectd -> components[dIndex] += vects -> components[dIndex];
      dIndex--;
   }
}

void VectorSub(vector *vectd, vector *vects)
{
   int   dIndex;

   if (vectd -> dimensions != vects -> dimensions)
      return;

   dIndex = vectd -> dimensions - 1;
   while (dIndex >= 0) {
      vectd -> components[dIndex] -= vects -> components[dIndex];
      dIndex--;
   }
}

void VectorNeg(vector *vect)
{
   int   dIndex;

   dIndex = vect -> dimensions - 1;
   while (dIndex >= 0) {
      vect -> components[dIndex] = -(vect -> components[dIndex]);
      dIndex--;
   }
}

void VectorSquare(vector *vect)
{
   int   dIndex;

   dIndex = vect -> dimensions - 1;
   while (dIndex >= 0) {
      vect -> components[dIndex] *= vect -> components[dIndex];
      dIndex--;
   }
}


void VectorMultiplyScalar(vector *vect, double scalar)
```

```
{
   int   dIndex,
      dimensions;       /* for speedup */

   dimensions = vect -> dimensions;
   dIndex = 0;
   while (dIndex < dimensions) {
      vect -> components[dIndex] *= scalar;
      dIndex++;
   }
}

void VectorDivideScalar(vector *vect, double scalar)
{
   VectorMultiplyScalar(vect, 1.0 / scalar);
}


vector *VectorMakeCopy(vector *vect)
{
   vector   *newVect;
   int   dIndex;     /* index into components (across dimensions) */

   if (vect == NULL)
      return NULL;

   if (!(newVect = VectorAllocate(vect -> dimensions)))
      return NULL;

   dIndex = 0;
   while (dIndex < vect -> dimensions) {
      newVect -> components[dIndex] = vect -> components[dIndex];
      dIndex++;
   }

   return newVect;
}


vector *VectorMakeHomogeneous(vector *vect)
{
   vector   *homVect;
   int      dIndex;     /* index into components (across dimensions) */

   if (vect == NULL)
      return NULL;

   if (!(homVect = VectorAllocate(vect -> dimensions + 1)))
      return NULL;

   dIndex = 0;
   while (dIndex < vect -> dimensions) {
      homVect -> components[dIndex] = vect -> components[dIndex];
      dIndex++;
   }

   homVect -> components[vect -> dimensions] = 1.0;

   return homVect;
}


int VectorMaxComponent(vector *vect)
{
   int   indexMax,
      cIndex;
   double   maxComponent;

   maxComponent = vect -> components[0];
   indexMax = 0;
   cIndex = 1;
   while (cIndex < vect -> dimensions) {
      if (vect -> components[cIndex] > maxComponent) {
         maxComponent = vect -> components[cIndex];
         indexMax = cIndex;
      }
      cIndex++;
   }

   return indexMax;
}
```

```
bool VectorIdentical(vector *vect1, vector *vect2)
{
   int   cIndex;

   if (vect1 -> dimensions != vect2 -> dimensions)
      return FALSE;

   cIndex = 0;
   while (cIndex < vect1 -> dimensions) {
      if (vect1 -> components[cIndex] != vect2 -> components[cIndex])
         return FALSE;
      cIndex++;
   }

   return TRUE;
}

double VectorManhattanEntropy(vector *vect1, vector *vect2)
{
/*
 *    H = -Sum(d)[fd * logn(fd)]
 *
 *              |ad - bd|
 *    fd = -------------------
 *          Sum(d)[|ad - bd|]
 *
 */
   double   mDistSum, fd, hSum;
   int   dIndex;

   if (vect1 -> dimensions != vect2 -> dimensions)
      return FP_SNAN;

   mDistSum = 0.0;
   for (dIndex = 0; dIndex < vect1 -> dimensions; dIndex++)
      mDistSum += ABS(vect1 -> components[dIndex] - vect2 -> components[dIndex]);

   if (mDistSum == 0.0)
      return 0.0;

   hSum = 0.0;
   for (dIndex = 0; dIndex < vect1 -> dimensions; dIndex++) {
      fd = ABS(vect1 -> components[dIndex] - vect2 -> components[dIndex]) / mDistSum;
      if (fd != 0.0) hSum -= fd * log(fd);
   }

   return hSum;
}


double **VectorCorrelation(vector *vect, int *matrixSize)
{
   double   **matrix;
   int   xIndex, yIndex;
   double   component;

   *matrixSize = vect -> dimensions;
   if (!Allocate2DArray((char ***) &matrix, *matrixSize, *matrixSize, sizeof(double)))
      return NULL;

   for (yIndex = 0; yIndex < *matrixSize; yIndex++) {
      component = vect -> components[yIndex];
      for (xIndex = 0; xIndex < *matrixSize; xIndex++)
         matrix[xIndex][yIndex] = component * vect -> components[xIndex];
   }

   return matrix;
}

/* --- END of vector_utils.c --- */
```

9.3.10 vector_read

Purpose	Provides utilities for reading and writing vectors from file streams
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	14th September, 1999
Modified	16th September, 1999
Module version	0.02

Other required modules	StdDefs, Vector_Utils, TokenLst, TokScan

vector_read.h

```
/* Vector read: utilities for reading and writing a vector
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 * Date: 14th September, 1999
 * Modified: 16th September, 1999
 * Version: 0.02
 */

#ifndef  __vector_read_h
#define  __vector_read_h

/* -- required modules -- */

#include "vector_utils.h"
#include "stddefs.h"
#include <stdio.h>


/* -- vector reading functions -- */

/* -- Function VectorReadLine
 * Pre: 'input' is a valid file opened for reading
 *      'inFile' is an allocated bool
 * Post: (a vector was read until EOL and was allocated and returned) ||
 *       (there was a problem with allocation && NULL was returned)
 *       'inFile' == (we have not reached EOF)
 */
vector *VectorReadLine(FILE *input, bool *inFile);

/* -- Function VectorReadComponents
 * Pre: 'input' is a valid file openend for reading
 *      'vect' is a valid allocated vector
 *      'inFile' is an allocated bool
 * Post: an entire line of 'input' was read; only the number
 *       of components in 'vect' were assigned; the rest of
 *       the line was discarded
 *       'inFile' == (we have not reached EOF)
 */
void VectorReadComponents(FILE *input, vector *vect, bool *inFile);

/* -- Function VectorWrite
 * Pre: 'output' is a file opened for writing
 *      'vect' is a valid allocated vector
 * Post: 'vect' has been written to 'output'
 * Note: 'vect's components are separated by tabs (\t)
 */
void VectorWrite(FILE *output, vector *vect);

#endif   /* __vector_read_h */

/* --- END of vector_read.h --- */
```

vector_read.c

```
/* vector read: utilities for reading a vector
 *
 * SEE vector_read.h for details
 */


/* -- vector read includes -- */

#include "vector_utils.h"
#include "stddefs.h"
#include <stdio.h>
#include "tokenlst.h"
#include "tokscan.h"
#include <stdlib.h>


/* -- internal structures -- */

/* -- vector read functions -- */

vector *VectorReadLine(FILE *input, bool *inFile)
```

```
{
   tokenList   *tokens, *tindex;
   vector      *vect;
   int      dimensions, compIndex;

   *inFile = !feof(input);

   tokens = ReadLineTokens(input, inFile);

   if (tokens == NULL) return NULL;    /* nothing on the line */

   dimensions = TokenListLength(tokens);
   if ((vect = VectorAllocate(dimensions)) == NULL) {
      DestroyTokenList(tokens);
      return NULL;
   }

   tindex = tokens;
   compIndex = 0;
   while ((compIndex < dimensions) && (tindex != NULL)) {
      vect -> components[compIndex] = atof(tindex -> token);
      tindex = tindex -> NEXT;
      compIndex++;
   }

   DestroyTokenList(tokens);

   return vect;
}


void VectorReadComponents(FILE *input, vector *vect, bool *inFile)
{
   tokenList   *tokens, *tindex;
   int      dimensions, compIndex;

   *inFile = !feof(input);

   tokens = ReadLineTokens(input, inFile);

   if (tokens == NULL) return;      /* nothing on the line */

   dimensions = vect -> dimensions;

   tindex = tokens;
   compIndex = 0;
   while ((compIndex < dimensions) && (tindex != NULL)) {
      vect -> components[compIndex] = atof(tindex -> token);
      tindex = tindex -> NEXT;
      compIndex++;
   }

   DestroyTokenList(tokens);
}

void VectorWrite(FILE *output, vector *vect)
{
   int   dIndex;        /* index into dimensions */

   dIndex = 0;
   while(dIndex < vect -> dimensions) {
      fprintf(output, "%3.2f\t", vect -> components[dIndex]);
      dIndex++;
   }
}

/* --- END of vector_read.c --- */
```

9.3.11 cluster

Purpose	A series of functions designed to represent and manipulate clusters of vectors.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	14th September, 1999
Modified	21st November, 2000
Module version	0.05
Notes	Essentially a linked-list of vectors.

Other required modules	StdDefs, Vector_Utils, Vector_Read

cluster.h

```
/* cluster: routines for dealing with clusters of vectors
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         QUT MLRC LPG Semester 2 1999
 * Date: 14th September, 1999
 * Modified: 21st November, 2000
 * Version: 0.05
 */

#ifndef  __cluster_h
#define  __cluster_h

/* -- required modules -- */

#include <stdio.h>
#include "vector_utils.h"
#include "stddefs.h"

/* -- Cluster structure -- */

struct clusterTag {
   vector            *vect;
   struct clusterTag *next;
};

typedef struct clusterTag cluster;


/* -- Cluster functions -- */

/* -- Function ClusterNew
 * Pre: TRUE
 * Post: (a new empty cluster was returned)
 * Note: a new cluster is always empty
 */
cluster  *ClusterNew(void);


/* -- Function ClusterDestroy
 * Pre: 'clust' is a valid allocated cluster
 * Post: 'clust' was destroyed && a NULL cluster was returned
 */
cluster  *ClusterDestroy(cluster *clust);


/* -- Function ClusterAddVector
 * Pre: 'clust' is a valid allocated cluster
 *      'vect' is a valid allocated vector
 * Post: a cluster containing both 'clust' and 'vect' was returned
 */
cluster  *ClusterAddVector(cluster *clust, vector *vect);


/* -- Function ClusterRemoveVector
 * Pre: 'clust' is a valid allocated cluster
 *      'vect' is a valid allocated vector
 * Post: a cluster with 'vect' removed was returned
 * Note: if two instances of 'vect' exist in 'clust', only one will be removed
 */
cluster *ClusterRemoveVector(cluster *clust, vector *vect);


/* -- Function ClusterMerge
 * Pre: 'clust1' and 'clust2' are valid allocated clusters
 * Post: a cluster containing both 'clust1' and 'clust2' was returned
 */
cluster *ClusterMerge(cluster *clust1, cluster *clust2);


/* -- Function ClusterCentroid
 * Pre: 'clust' is a valid allocated non-empty cluster
 * Post: the centroid of 'clust' was calcluated and returned
 * Note: this function allocates a new vector, which you
 *       MUST destroy
 */
```

```
vector   *ClusterCentroid(cluster *clust);


/* -- Function ClusterSize
 * Pre: 'clust' is a valid allocated cluster
 * Post: the number of vectors in 'clust' was returned
 */
int   ClusterSize(cluster *clust);


/* -- Function ClusterIsEmpty
 * Pre: 'clust' is a valid allocated cluster
 * Post: returned (ClusterSize(clust) == 0)
 */
bool  ClusterIsEmpty(cluster *clust);


/* -- Function ClusterReadFromFile
 * Pre: 'inputFile' is a vector file opened for reading
 * Post: All the vectors from 'inputFile' have been read and added to a cluster
 *       which is subsequently returned.  If no vectors could be read, an
 *       empty cluster is returned.
 * Note: This relies on VectorReadLine.  See vector_read.h for details
 */
cluster *ClusterReadFromFile(FILE *inputFile);


/* -- Function ClusterCheckDimensionality
 * Pre: 'clust' is a valid allocated cluster
 * Post: (all vectors in 'clust' have the same dimensions && TRUE was returned) ||
 *       (at least one vector had a different number of dimensions && FALSE was returned)
 */
bool ClusterCheckDimensionality(cluster *clust);


/* -- Function ClusterVectorIsIn
 * Pre: 'clust' is a valid allocated cluster
 *      'vect' is a valid allocated vector
 * Post: ('vect' is in 'clust' && TRUE was returned) ||
 *       ('vect' was not found in 'clust' && FALSE was returned)
 */
bool ClusterVectorIsIn(cluster *clust, vector *vect);


/* -- Function ClusterMakeCopy
 * Pre: 'clust' is the cluster to copy
 * Post: a copy of 'clust' was made and returned
 */
cluster *ClusterMakeCopy(cluster *clust);

#endif   /* __cluster_h */

/* --- END of cluster.h --- */
```

cluster.c

```
/* cluster : routines for dealing with clusters of vectors
 *
 * SEE cluster.h for details
 */

#include "cluster.h"
#include "vector_utils.h"
#include "vector_read.h"
#include "stddefs.h"
#include <stdlib.h>


cluster *ClusterAlloc(void)
{
   cluster  *clust;

   if ((clust = (cluster *) malloc(sizeof(cluster))) == NULL)
      return NULL;

   clust -> vect = NULL;
   clust -> next = NULL;

   return clust;
}
```

```
/* -- Cluster functions -- */

bool ClusterIsEmpty(cluster *clust)
{
   return (clust == NULL);
}

cluster *ClusterNew(void)
{
   return NULL;
}

cluster *ClusterDestroy(cluster *clust)
{
   if (ClusterIsEmpty(clust))
      return NULL;

   clust -> next = ClusterDestroy(clust -> next);
   VectorDeallocate(clust -> vect);
   free(clust);
   return NULL;
}

cluster *ClusterAddVector(cluster *clust, vector *vect)
{
   cluster  *newCluster;

   if (vect == NULL)
      return clust;

   if ((newCluster = ClusterAlloc()) == NULL) {
      ClusterDestroy(clust);
      return NULL;
   }

   newCluster -> vect = vect;
   newCluster -> next = clust;

   return newCluster;
}

cluster *ClusterMerge(cluster *clust1, cluster *clust2)
{
   cluster  *cIndex;

   if (ClusterIsEmpty(clust1))
      return clust2;

   if (ClusterIsEmpty(clust2))
      return clust1;

   cIndex = clust1;
   while (cIndex -> next != NULL)
      cIndex = cIndex -> next;

   cIndex -> next = clust2;
   return clust1;
}

vector *ClusterCentroid(cluster *clust)
{
   vector   *vectSum;
   cluster  *cIndex;    /* index into cluster */

   if (ClusterIsEmpty(clust))
      return NULL;

   if ((vectSum = VectorZero(clust -> vect -> dimensions)) == NULL)
      return NULL;

   cIndex = clust;
   while (cIndex != NULL) {
      VectorSum(vectSum, cIndex -> vect);
      cIndex = cIndex -> next;
   }

   VectorDivideScalar(vectSum, (double) ClusterSize(clust));

   return vectSum;
}
```

```
int ClusterSize(cluster *clust)
{
   int   size;
   cluster  *cIndex;

   if (ClusterIsEmpty(clust))
      return 0;

   size = 1;
   cIndex = clust;
   while (cIndex -> next != NULL) {
      size++;
      cIndex = cIndex -> next;
   }

   return size;
}

cluster *ClusterReadFromFile(FILE *inputFile)
{
   cluster  *clust = ClusterNew();
   bool     inFile;

   inFile = TRUE;
   while(inFile) {
      clust = ClusterAddVector(clust, VectorReadLine(inputFile, &inFile));
   }
   return clust;
}

cluster *ClusterRemoveVector(cluster *clust, vector *vect)
{
   cluster  *cIndex,
            *cTemp;

   if (vect == NULL)
      return clust;

   if (clust == NULL)
      return clust;

   if (VectorIdentical(clust -> vect, vect)) {
      cTemp = clust -> next;
      clust -> next = NULL;
      clust = ClusterDestroy(clust);
      return cTemp;
   }

   cIndex = clust;
   while (cIndex -> next != NULL) {
      if (VectorIdentical(cIndex -> next -> vect, vect)) {
         cTemp = cIndex -> next;
         cIndex -> next = cIndex -> next -> next;
         cTemp -> next = NULL;
         cTemp = ClusterDestroy(cTemp);
         return clust;
      }
      cIndex = cIndex -> next;
   }

   return clust;
}


bool ClusterVectorIsIn(cluster *clust, vector *vect)
{
   cluster  *cIndex;

   if (vect == NULL)
      return FALSE;

   if (clust == NULL)
      return FALSE;

   cIndex = clust;
   while (cIndex != NULL) {
      if (VectorIdentical(cIndex -> vect, vect)) {
         return TRUE;
      }
      cIndex = cIndex -> next;
   }
```

```
   return FALSE;
}


bool ClusterCheckDimensionality(cluster *clust)
{
   int      dimensions;              /* dimension to check against */
   cluster  *cIndex;                 /* index into cluster */

   if (clust == NULL) return TRUE; /* empty cluster, so we pass */

   dimensions = clust -> vect -> dimensions;
   cIndex = clust -> next;         /* the first one passes by definition */

   while (cIndex != NULL) {
      if (cIndex -> vect -> dimensions != dimensions)
         return FALSE;

      cIndex = cIndex -> next;
   }

   return TRUE;
}


cluster *ClusterMakeCopy(cluster *clust)
{
   cluster  *newClust;

   if (clust == NULL)
      return NULL;

   newClust = NULL;
   newClust = ClusterAddVector(newClust, VectorMakeCopy(clust -> vect));
   newClust -> next = ClusterMakeCopy(clust -> next);

   return newClust;
}


/* --- END of cluster.c --- */
```

9.3.12 corr_matrix

Purpose	Implementation of the second-order (correlation matrix) distance measure.
Author	Dylan Muir (dr.muir@student.qut.edu.au)
Date	5th November, 1999
Modified	15th December, 2000
Module version	0.21
Notes	This is by far the most complicated module in the repertoire. I **strongly** suggest reading the implementation description and other notes in section 2.2 of this report, in order to understand the second-order distance measure as well as the correlation matrix cluster representation.
Other required modules	2darray, Vector_Utils, Vector_Read, cluster, gauss
Acknowledgements	Lipson, H., Siegelmann, H. T., **High Order Shape Neurons for Data Structure Decomposition**

corr_matrix.h

```
/* Corr_matrix -- correlation maxtrix implementation for 2nd order clustering
 *
 * Author: Dylan Muir (dr.muir@student.qut.edu.au)
 *         (c) QUT MLRC Semester 2 1999
 *         (c) QUT MLRC Semester 1 2000
 *         (c) QUT MLRC Semester 2 2000
 * Date: 5th November, 1999
 * Modified: 15th December, 2000 (Post-GST)
 * Version: 0.21
 *
```

```
 * Aknowledgements: Lipson, H., Siegelmann, H. T., "High Order Shape Neurons for Data Structure
Decomposition"
 *
 * Uses: Vector_utils, Cluster, Stddefs
 */

#ifndef  __corr_matrix
#define  __corr_matrix

/* -- Correlation matrix includes -- */

#include <stdlib.h>
#include "cluster.h"
#include "vector_utils.h"
#include "stddefs.h"

#ifdef   WIN32
   #include <float.h>
   #define  FP_SNAN  _FPCLASS_SNAN
#endif

#define  MIN_VECTORS_IN_CLUSTER  8     /* approximate cluster with a dummy until we have x vectors
*/
#define  INITIAL_DISTANCE        0.1   /* for an approximate cluster, how far away  */
                                       /* from the centroid to put the dummy points */

typedef struct CMatrixTag {
   int   dimensions,       /* Dimensionality of vectors */
         num_vectors;      /* Used for normalising MATRIX */
   double   **MATRIX,      /* Covariance matrix [dim x dim] */
            **INVERSE,     /* Correlation matrix inverse [dim + 1 x dim + 1] */
            avCorrelation, /* Average correlation from all vectors within cluster */
            stddevCorr;    /* Standard deviation correlation for all vectors within cluster */
   bool     singular,      /* Matrix is singular */
            validAverage,  /* Average correlation is valid (otherwise calculate it) */
            isDummy;       /* Is this matrix a dummy cluster? */
   vector   *centroid;     /* Euclidean centroid of cluster */
   cluster  *vector_set;   /* Set of vectors comprising cluster */
} *corrMatrix;

/* Notes:
 *
 * The full (conceptual) correlation matrix contains vectors in homogeneous
 * form, and augments the correlation terms with what becomes effectively the
 * cluster centroids along the right side and on the bottom.  The full matrix
 * is also in normalised form.
 *
 * Rh = Sum (cluster) [X Trans(X)]   <-- Trans == transpose (i.e. vector product)
 *
 *    = Sum [a][a b c 1]
 *          [b]
 *          [c]
 *          [1]  <-- Homogeneous form adds this to every pattern vector
 *
 *    = Sum (j) [a2 ab ac | a] = Rh(j)
 *              [ba b2 bc | b]
 *              [ca cb c2 | c]
 *              [---------+  ]
 *              [ a  b  c   1]
 *
 * This matrix is then normalised by dividing by size(cluster)
 * The correlation of a vector with (cluster) is given by finding the
 * product of X * the inverse of Rh * Trans(X)
 */

/* -- Correlation matrix functions -- */

/* -- Function CMatrixCreate
 * Pre: 'dimensions' > 0
 * Post: (A new empty correlation matrix was created and returned) ||
 *       (There was a problem with allocation && NULL was returned)
 */
corrMatrix CMatrixCreate(int dimensions);

/* -- Function CMatrixDestroy
 * Pre: 'matrix' is a valid allocated correlation matrix
 * Post: 'matrix' was destroyed && a null matrix was returned
 */
corrMatrix CMatrixDestroy(corrMatrix matrix);

/* -- Function CMatrixWipe
 * Pre: 'matrix' is a valid allocated matrix
```

```
 * Post: 'matrix' was cleared so that it contains no vectors
 */
void CMatrixWipe(corrMatrix matrix);

/* -- Function CMatrixCreateFromCluster
 * Pre: 'clust' is a valid cluster
 * Post: (a new matrix was created from 'clust' and returned) ||
 *       (there was a problem with allocation && NULL was returned)
 */
corrMatrix CMatrixCreateFromCluster(cluster *clust);

/* -- Function CMatrixCreateFromVectorSet
 * Pre: 'vectorSet' is an array of allocated vectors
 *      'numVectors' is the number of vectors in 'vectorSet'
 * Post: (a new matrix was created from COPIES of the vectors in 'vectorSet' and returned) ||
 *       (there was a problem with allocation && NULL was returned)
 */
corrMatrix CMatrixCreateFromVectorSet(vector *vectorSet[], int numVectors);

/* -- Function CMatrixMakeCopy
 * Pre: 'matrix' is a valid allocated matrix
 * Post: (a new matrix was copied from 'matrix and returned) ||
 *       (there was a problem with allocation && NULL was reurned)
 */
corrMatrix CMatrixMakeCopy(corrMatrix matrix);

/* -- Function CMatrixAddVector
 * Pre: 'matrix' is a valid allocated matrix
 *      'vect' is a valid allocated vector
 * Post: ('vect' was added to 'matrix' && TRUE was returned) ||
 *       ('vect' could not be added to 'matrix' && FALSE was returned)
 * Note: This functions adds a COPY of 'vect'
 */
bool CMatrixAddVector(corrMatrix matrix, vector *vect);

/* -- Function CMatrixDeleteVector
 * Pre: 'matrix' is a valid allocated matrix
 *      'vect' is a valid allocated vector
 * Post: ('vect' was in 'matrix' && 'vect' was removed from 'matrix' && TRUE was returned) ||
 *       ('vect' was not in 'matrix' && FALSE was returned)
 */
bool CMatrixDeleteVector(corrMatrix matrix, vector *vect);

/* -- Function CMatrixVectorIsIn
 * Pre: 'matrix' is a valid allocated matrix
 *      'vect' is a valid allocated vector
 * Post: ('vect' exists in 'matrix' && TRUE was returned) ||
 *       ('vect' does not exist in 'matrix' && FALSE was returned)
 */
bool CMatrixVectorIsIn(corrMatrix matrix, vector *vect);

/* -- Function CMatrixIsEmpty
 * Pre: 'matirx' is a valid allocated matrix
 * Post: ('matrix' contains no vectors & TRUE was returned) ||
 *       ('matrix' contains at least one vector && FALSE was returned)
 */
bool CMatrixIsEmpty(corrMatrix matrix);

/* -- Function CMatrixCorrelationWithVect
 * Pre: 'matrix' is a valid non-empty allocated matrix
 *      'vect' is a valid allocated vector
 * Post: If 'matrix' contains at least three vectors, it will be treated as
 *       a cluster and the correlation will be calculated.  If the matrix contains
 *       less than three vectors, the Euclidean distance to the centroid (a spherical
 *       measure) will be returned instead.  If 'matrix' is singular, FP_SNAN will be
 *       returned.
 */
double CMatrixCorrelationWithVect(corrMatrix matrix, vector *vect);

/* -- Function CMatrixNormedCorrWithVect
 * Pre: 'matrix' is a valid non-empty allocated matrix
 *      'vect' is a valid allocated vector
 * Post: Returns the normalised correlation from 'vect' -> 'matrix'
 */
double CMatrixNormedCorrWithVect(corrMatrix matrix, vector *vect);

/* -- Function CMatrixNormWithVector
 * Pre: 'matrix' is a valid non-empty allocated matrix
 *      'vect' is a valid allocated vector
 * Post: The euclidean norm of the vector product of 'vect' and 'matrix' was
 *       returned (See top of file for description).  If 'matrix' is singular, FP_SNAN
 *       will be returned.
```

```
 */
double CMatrixNormWithVector(corrMatrix matrix, vector *vect);

/* -- Function CMatrixGetConceptualMatrix
 * Pre: 'matrix' is a valid allocated matrix
 *      'matrixSize' points to an allocated integer
 * Post: The conceptual (real) correlation matrix is returned as a 2D
 *       double array.  The matrix is square; the size is placed in 'matrixSize'
 *       NULL is returned if allocation was not possible.  In this case,
 *       'matrixSize' is undefined
 */
double **CMatrixGetConceptualMatrix(corrMatrix matrix, int *matrixSize);

/* -- Function CMatrixWrite
 * Pre: 'output' is a file opened for writing
 *      'matrix' is a valid allocated matrix
 * Post: the matrix of 'matrix' was written to output in a way that can be read with
 *        the CMatrixRead function
 */
void CMatrixWrite(FILE *output, corrMatrix matrix);

/* -- Function CMatrixRead
 * Pre: 'input' is a file opened for writing
 * Post: (a new matrix was read from 'input' and returned) &&
 *       (there was a problem reading the file && NULL was returned)
 */
corrMatrix CMatrixRead(FILE *input);

/* -- Function CMatrixWriteConceptual
 * Pre: 'output' is a file opened for writing
 *      'matrix' is a valid allocated matrix
 * Post: the conceptual matrix of 'matrix' was written to output
 */
void CMatrixWriteConceptual(FILE *output, corrMatrix matrix);

/* -- Function CMatrixWriteMatlab
 * Pre: 'output' is a file opened for writing
 *      'matrix' is a valid allocated matrix
 * Post: the conceptual matrix of 'matrix' was written to output in a format
 *       that is easy to copy into matlab
 */
void CMatrixWriteMatlab(FILE *output, corrMatrix matrix);

/* -- Function CMatrixRMSCorrelation
 * Pre: 'matrix' is a valid allocated matrix
 * Post: the RMS correlation over all vectors in 'matrix' was computed and returned
 */
double CMatrixRMSCorrelation(corrMatrix matrix);

/* -- Fnuction CMatrixAvCorrWithinCluster
 * Pre: 'matrix' is a valid allocated matrix with >= 3 vectors
 * Post: The average correlation from all vectors within cluster to the matrix was
 *       calculated and returned
 */
double CMatrixAvCorrWithinCluster(corrMatrix matrix);

/* -- Fnuction CMatrixStdDevCorrWithinCluster
 * Pre: 'matrix' is a valid allocated matrix with >= 3 vectors
 * Post: The standard deviation from all vectors within cluster to the matrix was
 *       calculated and returned
 */
double CMatrixStdDevCorrWithinCluster(corrMatrix matrix);

/* -- Function CMatrixCentroid
 * Pre: 'matrix' is a valid allocated matrix
 * Post: 'matrix's centroid was returned as an EXISTING vector: must be duplicated
 */
vector *CMatrixCentroid(corrMatrix matrix);

/* -- Function CMatrixMerge
 * Pre: 'dest' is a valid corrMatrix
 *      'source' is a valid corrMatrix
 * Post: (All the vectors in 'source' were COPIED into 'dest' && TRUE was returned) ||
 *       (there was a problem with allocation && FALSE was returned)
 */
bool CMatrixMerge(corrMatrix dest, corrMatrix source);

/* -- Function CMatrixUseDummy
 * Pre: 'matrix' is a valid matrix
 * Post: ('matrix' doesn't meet the criteria for a cluster && returned TRUE) ||
 *       (returned FALSE)
 */
```

```
bool CMatrixUseDummy(corrMatrix matrix);


#endif   __corr_matrix

/* --- END of corr_matrix.h --- */
```

corr_matrix.c

```
/* Corr_matrix -- correlation matrix implementation
 *
 * SEE corr_matrix.h for details
 */


/* -- Correlation Matrix includes -- */

#include <stdlib.h>
#ifdef   UNIX
   #include <ieeefp.h>
   #define  FP_MAX   FP_PINF
#endif
#ifdef   WIN32
   #include <float.h>
   #define  FP_SNAN  _FPCLASS_SNAN
   #define  FP_MAX   DBL_MAX
#endif
#include <memory.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include "corr_matrix.h"
#include "2darray.h"
#include "vector_utils.h"
#include "vector_read.h"
#include "cluster.h"
#include "gauss.h"


/* -- Correlation Matrix defines -- */

/* --- SQR definded in nrutil.h --- */
#define  SQR(x)   ((x) * (x))
/* -------------------------------- */


/* -- Correlation Matrix help function definitions -- */

/* -- Function AddCorrelation
 * Pre: 'matrix' is a valid allocated matrix
 *      'correlation' is the partial correlation matrix for a single vector
 * Post: the components of 'correlation' were added to 'matrix'
 */
void AddCorrelation(corrMatrix matrix, double **correlation);

/* -- Function SubtractCorrelation
 * Pre: 'matrix' is a valid allocated matrix
 *      'correlation is the partial correlation matrix for a single vector
 * Post: the components of 'correlation' were subtracted from 'matrix'
 */
void SubtractCorrelation(corrMatrix matrix, double **correlation);

/* -- Function CopyCorrelation
 * Pre: 'matrixd' is an allocated matrix
 *      'matrixs' is a valid allocated matrix to copy
 * Post: the componets of 'matrixs' were copied into 'matrixd'
 */
void CopyCorrelation(double **matrixd, double **matrixs, int dimensions);

/* -- Function UpdateInverse
 * Pre: 'matrix' is avalid allocated correlation matrix
 * Post: the inverse field of 'matrix' has been updated
 */
void UpdateInverse(corrMatrix matrix);

/* -- Function UpdateDummyInverse
 * Pre: 'matrix' is a valid allocated correlation matrix with less than
 *      'MIN_VECTORS_IN_CLUSTER' vectors
 * Post: the inverse field of 'matrix' was updated with an inverse based on the
 *       centroid of 'matrix', but with 'MIN_VECTORS_IN_CLUSTER' vectors.
 */
```

```
void UpdateDummyInverse(corrMatrix matrix);

/* -- Function MakeDummyCluster
 * Pre: 'matrix' is a valid allocated correlation matrix
 * Post: (A new cluster was created containing the dummy points used to make
 *        a dummy correlation matrix && the cluster was returned) ||
 *       (There was a problem making the cluster && NULL was returned)
 */
cluster *MakeDummyCluster(corrMatrix matrix);

/* -- Function CalculateAverage
 * Pre: 'matrix' is a valid allocated correlation matrix
 * Post: The 'avCorrelation' and 'stddevCorr' fields for 'matrix' are now valid &&
 *       'validAverage' == TRUE
 */
void CalculateAverage(corrMatrix matrix);


/* -- Function WriteHex
 * Pre: 'buffer' is an allocated text buffer that 'data' will be printed into in hex
 *      'buffer' must be at least 2 * 'numBytes' + 1 bytes long
 *      'data' is a pointer to the data region to output
 *      'numBytes' is the number of bytes of 'data' to print
 * Post: 'data' was printed in hex to 'buffer'
 */
void WriteHex(char buffer[], const unsigned char *data, size_t numBytes);

/* -- Function ReadHex
 * Pre: 'buffer' is an allocated text buffer that 'data' will be read from
 *      'data' is a pointer to the data region to output (pre-allocated)
 *      'numBytes' is the size of 'data' in bytes'
 * Post: 'data' was printed in hex to 'buffer'
 */
void ReadHex(const char buffer[], unsigned char *data, size_t numBytes);


/* -- Correlation Matrix functions -- */

corrMatrix CMatrixCreate(int dimensions)
{
   corrMatrix  matrix;
   int         xIndex, yIndex;

   if (!(matrix = (corrMatrix) malloc(sizeof(struct CMatrixTag))))
      return NULL;

   if (!Allocate2DArray((char ***) &(matrix -> MATRIX), dimensions, dimensions, sizeof(double))) {
      free(matrix);
      return NULL;
   }

   matrix -> dimensions = dimensions;
   matrix -> num_vectors = 0;
   matrix -> centroid = NULL;
   matrix -> INVERSE = NULL;
   matrix -> validAverage = FALSE;
   matrix -> singular = FALSE;
   matrix -> vector_set = ClusterNew();
   matrix -> isDummy = FALSE;

   for (yIndex = 0; yIndex < dimensions; yIndex++)
      for (xIndex = 0; xIndex < dimensions; xIndex++)
         matrix -> MATRIX[xIndex][yIndex] = 0.0;

   return matrix;
}


corrMatrix CMatrixDestroy(corrMatrix matrix)
{
   CMatrixWipe(matrix);
   Deallocate2DArray((char ***) &(matrix -> MATRIX), matrix -> dimensions);
   if (matrix -> INVERSE)
      Deallocate2DArray((char ***) &(matrix -> INVERSE), matrix -> dimensions + 1);
   free(matrix);
   return NULL;
}


void CMatrixWipe(corrMatrix matrix)
{
   int   xIndex, yIndex;
```

```
   if (matrix -> num_vectors == 0)
      return;

   if (matrix -> centroid) matrix -> centroid = VectorDeallocate(matrix -> centroid);
   matrix -> vector_set = ClusterDestroy(matrix -> vector_set);
   matrix -> num_vectors = 0;

   for (yIndex = 0; yIndex < matrix -> dimensions; yIndex++)
      for (xIndex = 0; xIndex < matrix -> dimensions; xIndex++)
         matrix -> MATRIX[xIndex][yIndex] = 0.0;

   if (matrix -> INVERSE) {
      Deallocate2DArray((char ***) &(matrix -> INVERSE), matrix -> dimensions + 1);
      matrix -> INVERSE = NULL;
   }
   matrix -> singular = FALSE;
   matrix -> validAverage = FALSE;
   matrix -> isDummy = FALSE;
}


corrMatrix CMatrixCreateFromCluster(cluster *clust)
{
   corrMatrix  matrix;
   cluster     *cIndex;

   if (!ClusterCheckDimensionality(clust))
      return NULL;

   if (!(matrix = CMatrixCreate(clust -> vect -> dimensions)))
      return NULL;

   cIndex = clust;
   while (cIndex != NULL) {
      if (!CMatrixAddVector(matrix, cIndex -> vect)) {
         matrix = CMatrixDestroy(matrix);
         return NULL;
      }
      cIndex = cIndex -> next;
   }

   return matrix;
}


corrMatrix CMatrixMakeCopy(corrMatrix matrix)
{
   corrMatrix  copy;

   copy = CMatrixCreate(matrix -> dimensions);

   copy -> num_vectors = matrix -> num_vectors;
   copy -> centroid = VectorMakeCopy(matrix -> centroid);
   copy -> vector_set = ClusterMakeCopy(matrix -> vector_set);
   copy -> singular = matrix -> singular;
   CopyCorrelation(copy -> MATRIX, matrix -> MATRIX, copy -> dimensions);
   CopyCorrelation(copy -> INVERSE, matrix -> INVERSE, copy -> dimensions + 1);
   if (matrix -> validAverage) {
      copy -> avCorrelation = matrix -> avCorrelation;
      copy -> stddevCorr = matrix -> stddevCorr;
      copy -> validAverage = TRUE;
   }
   copy -> isDummy = matrix -> isDummy;

   return copy;
}


bool CMatrixAddVector(corrMatrix matrix, vector *vect)
{
   double   **correlation;
   int   dimensions;

   vector   *vectNorm;

   dimensions = vect -> dimensions;
   if (dimensions != matrix -> dimensions)
      return FALSE;

   vectNorm = VectorMakeCopy(vect);
```

```
   if (!(correlation = VectorCorrelation(vect, &dimensions)))
      return FALSE;

   AddCorrelation(matrix, correlation);
   matrix -> vector_set = ClusterAddVector(matrix -> vector_set, VectorMakeCopy(vect));
   matrix -> validAverage = FALSE;
   if (matrix -> centroid) {
      VectorDeallocate(matrix -> centroid);
      matrix -> centroid = NULL; /*ClusterCentroid(matrix -> vector_set);*/
   }

   Deallocate2DArray((char ***) &correlation, dimensions);
   if (matrix -> INVERSE) {
      Deallocate2DArray((char ***) &(matrix -> INVERSE), matrix -> dimensions + 1);
      matrix -> INVERSE = NULL;
      matrix -> singular = FALSE;
   }

   matrix -> num_vectors++;

   return TRUE;
}

bool CMatrixDeleteVector(corrMatrix matrix, vector *vect)
{
   double   **correlation;
   int   dimensions;

   if (!(correlation = VectorCorrelation(vect, &dimensions)))
      return FALSE;

   if (!ClusterVectorIsIn(matrix -> vector_set, vect))
      return FALSE;

   SubtractCorrelation(matrix, correlation);
   matrix -> vector_set = ClusterRemoveVector(matrix -> vector_set, vect);
   if (matrix -> centroid) {
      VectorDeallocate(matrix -> centroid);
      matrix -> centroid = NULL; /*ClusterCentroid(matrix -> vector_set);*/
   }
   matrix -> validAverage = FALSE;

   Deallocate2DArray((char ***) &correlation, dimensions);
   if (matrix -> INVERSE) {
      Deallocate2DArray((char ***) &(matrix -> INVERSE), matrix -> dimensions + 1);
      matrix -> INVERSE = NULL;
      matrix -> singular = FALSE;
   }

   matrix -> num_vectors--;
   return TRUE;
}


bool CMatrixVectorIsIn(corrMatrix matrix, vector *vect)
{
   return ClusterVectorIsIn(matrix -> vector_set, vect);
}


bool CMatrixIsEmpty(corrMatrix matrix)
{
   return matrix -> num_vectors == 0;
}

void printmatrix(double **matrix, int dimensions)
{
   int   iCol, iRow;

   if (matrix) {
      for (iRow = 0; iRow < dimensions; iRow++) {
         for (iCol = 0; iCol < dimensions; iCol++)
            printf("%12.8f  ", matrix[iRow][iCol]);
         printf("\n");
      }
   } else
      printf("{NULL}\n");
}

double CMatrixCorrelationWithVect(corrMatrix matrix, vector *vect)
{
   double   sum;
```

```
   int   dimensions,
         iCol, iRow;

   vector   *vectH;

   if (CMatrixIsEmpty(matrix))
      return FP_MAX;

   if (!matrix -> INVERSE)
      UpdateInverse(matrix);

   if (matrix -> singular)             /* Singular matrix; we can't calculate */
      return FP_SNAN;                   /* the correlation, so return NAN      */

   dimensions = matrix -> dimensions;

   vectH = VectorMakeHomogeneous(vect);

   sum = 0.0;
   for (iRow = 0; iRow < dimensions + 1; iRow++) {
      for (iCol = 0; iCol < dimensions + 1; iCol++)
         sum  +=  (vectH  ->  components[iRow])  *  (matrix  ->  INVERSE[iCol][iRow])  *  (vectH  ->
components[iCol]);
   }

   VectorDeallocate(vectH);

   return sum;
}


double CMatrixNormedCorrWithVect(corrMatrix matrix, vector *vect)
{
   if (CMatrixIsEmpty(matrix))
      return FP_MAX;

/* if (matrix -> num_vectors < 3) {
      if (!matrix -> centroid)
         matrix -> centroid = ClusterCentroid(matrix -> vector_set);

      return VectorEuclideanDist(vect, matrix -> centroid);
   }
*/
   if (!matrix -> validAverage) {
      CalculateAverage(matrix);
      matrix -> validAverage = TRUE;
   }

   return CMatrixCorrelationWithVect(matrix, vect) / matrix -> avCorrelation;
}

double CMatrixNormWithVector(corrMatrix matrix, vector *vect)
{
   double   *product,
            sum,
            num_vectors;
   int   dimensions,
         iCol, iRow;

   if (CMatrixIsEmpty(matrix))
      return FP_MAX;

   dimensions = matrix -> dimensions;
   num_vectors = (double) matrix -> num_vectors;
/*
   printf("Norming matrix:\n");
   printmatrix(matrix -> MATRIX, dimensions);
   printf("--------------\n");
   printmatrix(matrix -> INVERSE, dimensions);
   printf("With vector: ");
   VectorWrite(stdout, vect);
   printf("\n");
*/
   if (!matrix -> INVERSE)
      UpdateInverse(matrix);

   if (matrix -> singular)             /* Singular matrix, we can't calculate */
      return FP_SNAN;                   /* the 2-norm, so return NAN           */

   if (!(product = (double *) malloc((dimensions + 1) * sizeof(double))))
      return FP_SNAN;
```

```
   for (iRow = 0; iRow < dimensions + 1; iRow++) {
      sum = 0.0;
      for (iCol = 0; iCol < dimensions; iCol++)
         sum += (matrix -> INVERSE[iCol][iRow] * vect -> components[iCol]);
      sum += matrix -> INVERSE[dimensions][iRow]; /* * 1.0; */
      product[iRow] = sum;
   }

   sum = 0.0;
   for (iRow = 0; iRow < dimensions + 1; iRow++)
      sum += SQR(product[iRow]);

   sum = sqrt(sum);

   free(product);
   return sum;
}

/*
   product[dimensions] = 0.0;
   for (yIndex = 0; yIndex < dimensions; yIndex++) {
      sum = 0.0;
      for (xIndex = 0; xIndex < dimensions; xIndex++)
         sum  +=  (matrix  ->  MATRIX[xIndex][yIndex]  /  (double)  matrix  ->  num_vectors)  *  vect  ->
components[xIndex];
      sum += matrix -> centroid -> components[yIndex];
      product[yIndex] = sum;
      product[dimensions]  +=  matrix  ->  centroid  ->  components[yIndex]  /*  *  (double)  matrix  ->
num_vectors * vect -> components[yIndex];
   }

   product[dimensions] += 1.0;

   sum = 0.0;
   for (yIndex = 0; yIndex <= dimensions; yIndex++)
      sum += SQR(product[yIndex]); /* / product[dimensions]);

   sum += 1.0;
   sum = sqrt(sum);

   free(product);
   return sum;
}*/


double **CMatrixGetConceptualMatrix(corrMatrix matrix, int *matrixSize)
{
   double   **concepMatrix;
   int      iCol, iRow;

   *matrixSize = matrix -> dimensions + 1;

   if (!Allocate2DArray((char ***) &concepMatrix, *matrixSize, *matrixSize, sizeof(double)))
      return NULL;

   if (!matrix -> centroid)
      matrix -> centroid = ClusterCentroid(matrix -> vector_set);

   for (iRow = 0; iRow < *matrixSize - 1; iRow++) {
      for (iCol = 0; iCol < *matrixSize - 1; iCol++)
         concepMatrix[iCol][iRow] = matrix -> MATRIX[iCol][iRow] / (double) matrix -> num_vectors;
      concepMatrix[*matrixSize - 1][iRow] = matrix -> centroid -> components[iRow];
   }

   for (iCol = 0; iCol < *matrixSize - 1; iCol++)
      concepMatrix[iCol][*matrixSize - 1] = matrix -> centroid -> components[iCol];

   concepMatrix[*matrixSize - 1][*matrixSize - 1] = 1.0;

   return concepMatrix;
}


void CMatrixWrite(FILE *output, corrMatrix matrix)
{
   int   iRow, iCol,
         dimensions;
   char  buffer[50];
   vector   *centroid;

   dimensions = matrix -> dimensions;
```

```
   fprintf(output, "d: %d\n", dimensions);
   fprintf(output, "nv: %d\n", matrix -> num_vectors);

   if (!matrix -> validAverage)
      CalculateAverage(matrix);

   WriteHex(buffer, (char *) &(matrix -> avCorrelation), sizeof(double));
   fprintf(output, "ac: %s\n", buffer);
   WriteHex(buffer, (char *) &(matrix -> stddevCorr), sizeof(double));
   fprintf(output, "sdc: %s\n", buffer);
   fprintf(output, "sing: %d\n", matrix -> singular);

   centroid = CMatrixCentroid(matrix);

   fprintf(output, "c: ");

   for (iRow = 0; iRow < dimensions; iRow++) {
      WriteHex(buffer, (char *) &(centroid -> components[iRow]), sizeof(double));
      fprintf(output, "%s ", buffer);
   }
   fprintf(output, "\nMATRIX\n");

   for (iRow = 0; iRow < dimensions; iRow++) {
      for (iCol = 0; iCol < dimensions; iCol++) {
         WriteHex(buffer, (char *) &(matrix -> MATRIX[iRow][iCol]), sizeof(double));
         fprintf(output, "%s ", buffer);
      }
      fprintf(output, "\n");
   }
}

corrMatrix CMatrixRead(FILE *input)
{
   corrMatrix  newMatrix;
   int         iRow, iCol, dimensions;
   char        buffer[50];
   vector      *centroid;

   if (fscanf(input, " d: %d", &dimensions) == 0)
      return NULL;

   newMatrix = CMatrixCreate(dimensions);

   fscanf(input, " nv: %d", &(newMatrix -> num_vectors));
   newMatrix -> validAverage = TRUE;
   fscanf(input, " ac: %s", buffer);
   ReadHex(buffer, (char *) &(newMatrix -> avCorrelation), sizeof(double));
   fscanf(input, " sdc: %s", buffer);
   ReadHex(buffer, (char *) &(newMatrix -> stddevCorr), sizeof(double));

   fscanf(input, " sing: %d", &(newMatrix -> singular));

   if (!(centroid = VectorAllocate(dimensions))) {
      CMatrixDestroy(newMatrix);
      return NULL;
   }

   fscanf(input, " c: ");
   for (iRow = 0; iRow < dimensions; iRow++) {
      fscanf(input, "%s ", buffer);
      ReadHex(buffer, (char *) &(centroid -> components[iRow]), sizeof(double));
   }

   newMatrix -> centroid = centroid;

   if (!Allocate2DArray((char ***) &(newMatrix -> MATRIX), dimensions, dimensions, sizeof(double)))
{
      CMatrixDestroy(newMatrix);
      return NULL;
   }

   fscanf(input, " MATRIX ");
   for (iRow = 0; iRow < dimensions; iRow++) {
      for (iCol = 0; iCol < dimensions; iCol++) {
         fscanf(input, "%s ", buffer);
         ReadHex(buffer, (char *) &(newMatrix -> MATRIX[iRow][iCol]), sizeof(double));
      }
   }

   return newMatrix;
}
```

```
void CMatrixWriteMatlab(FILE *output, corrMatrix matrix)
{
   int    dimensions,
          iCol, iRow;
   double   **concepMatrix;

   if (!(concepMatrix = CMatrixGetConceptualMatrix(matrix, &dimensions))) {
      fprintf(stderr, "*** Error getting conceptual matrix.\n");
      return;
   }

   fprintf(output, "[");
   for (iRow = 0; iRow < dimensions; iRow++) {
      for (iCol = 0; iCol < dimensions; iCol++) {
         fprintf(output, "%f", concepMatrix[iCol][iRow]);
         if (iCol < dimensions - 1)
            fprintf(output, ",");
      }
      if (iRow < dimensions - 1)
         fprintf(output, ";");
   }
   fprintf(output, "]");
}

double CMatrixRMSCorrelation(corrMatrix matrix)
{
   double   corrSum;
   cluster  *cIndex;

   if (CMatrixIsEmpty(matrix))
      return 0.0;

   corrSum = 0.0;
   cIndex = matrix -> vector_set;
   while (cIndex != NULL) {
      corrSum += SQR(CMatrixCorrelationWithVect(matrix, cIndex -> vect));
      cIndex = cIndex -> next;
   }

/* corrSum /= (double) matrix -> num_vectors;*/
   return sqrt(corrSum);
}

double CMatrixAvCorrWithinCluster(corrMatrix matrix)
{
   if (!matrix -> validAverage)
      CalculateAverage(matrix);

   return matrix -> avCorrelation;
}


double CMatrixStdDevCorrWithinCluster(corrMatrix matrix)
{
   if (!matrix -> validAverage)
      CalculateAverage(matrix);

   return matrix -> stddevCorr;
}

vector *CMatrixCentroid(corrMatrix matrix)
{
   if (!matrix -> centroid)
      matrix -> centroid = ClusterCentroid(matrix -> vector_set);

   return matrix -> centroid;
}

bool CMatrixMerge(corrMatrix dest, corrMatrix source)
{
   cluster  *cIndex;

   if (source -> num_vectors == 0)
      return TRUE;

   cIndex = source -> vector_set;
   while (cIndex != NULL) {
      if (!CMatrixAddVector(dest, VectorMakeCopy(cIndex -> vect)))
         return FALSE;

      cIndex = cIndex -> next;
   }
```

```
   return TRUE;
}


bool CMatrixUseDummy(corrMatrix matrix)
{
   if (matrix -> num_vectors < MIN_VECTORS_IN_CLUSTER)
      return TRUE;
   else
      return FALSE;
}


/* -- Correlation Matrix helper functions -- */
void AddCorrelation(corrMatrix matrix, double **correlation)
{
   int   dimensions,
      xIndex, yIndex;

   dimensions = matrix -> dimensions;

   for (yIndex = 0; yIndex < dimensions; yIndex++)
      for (xIndex = 0; xIndex < dimensions; xIndex++)
         matrix -> MATRIX[xIndex][yIndex] += correlation[xIndex][yIndex];
}


void SubtractCorrelation(corrMatrix matrix, double **correlation)
{
   int   dimensions,
      xIndex, yIndex;

   dimensions = matrix -> dimensions;

   for (yIndex = 0; yIndex < dimensions; yIndex++)
      for (xIndex = 0; xIndex < dimensions; xIndex++)
         matrix -> MATRIX[xIndex][yIndex] -= correlation[xIndex][yIndex];
}

void CopyCorrelation(double **matrixd, double **matrixs, int dimensions)
{
   int   xIndex, yIndex;

   for (yIndex = 0; yIndex < dimensions; yIndex++)
      for (xIndex = 0; xIndex < dimensions; xIndex++)
         matrixd[xIndex][yIndex] = matrixs[xIndex][yIndex];
}

void UpdateInverse(corrMatrix matrix)
{
   double   **conceptual;
   int      dimINVERSE, dimConceptual;

   dimINVERSE  = matrix -> dimensions + 1;

   if (matrix -> INVERSE)
      Deallocate2DArray((char ***) &(matrix -> INVERSE), dimINVERSE);

   if (!CMatrixIsEmpty(matrix)) {
      if (!(matrix -> isDummy) && CMatrixUseDummy(matrix)) {     /* Less than x vectors, do dummy */
         UpdateDummyInverse(matrix);

      } else {
         if (!(conceptual = CMatrixGetConceptualMatrix(matrix, &dimConceptual)))
            return;

         if   (!Allocate2DArray((char   ***)   &(matrix   ->   INVERSE),   dimINVERSE,   dimINVERSE,
sizeof(double))) {
            fprintf(stderr, "--- 2DArray allocation failed for matrix inverse.\n");
            return;
         }

         CopyCorrelation(matrix -> INVERSE, conceptual, dimINVERSE);
         if (!GaussJordanInverse(conceptual, matrix -> INVERSE, dimINVERSE)) {
            /*fprintf(stderr, "--- Warning: Matrix was singular.\n");*/
            matrix -> singular = TRUE;
         }
         Deallocate2DArray((char ***) &conceptual, dimConceptual);
      }
   }
}
```

```
void UpdateDummyInverse(corrMatrix matrix)
{
   corrMatrix  dummyMatrix;
   int      numDimensions;
   cluster  *dummy, *clustIndex;

   numDimensions = matrix -> dimensions;
   dummyMatrix = CMatrixCreate(numDimensions);

   if (!matrix -> centroid)
      matrix -> centroid = ClusterCentroid(matrix -> vector_set);

   clustIndex = dummy = MakeDummyCluster(matrix);
   while (clustIndex != NULL) {
      CMatrixAddVector(dummyMatrix, clustIndex -> vect);
      clustIndex = clustIndex -> next;
   }

   dummyMatrix -> isDummy = TRUE;
   UpdateInverse(dummyMatrix);

   if (!(matrix -> INVERSE)) {
      Allocate2DArray(  (char ***) &(matrix -> INVERSE),
                        numDimensions + 1, numDimensions + 1,
                        sizeof(double));
   }

   CopyCorrelation(matrix -> INVERSE, dummyMatrix -> INVERSE, numDimensions + 1);

   CMatrixDestroy(dummyMatrix);
   ClusterDestroy(dummy);
}

cluster *MakeDummyCluster(corrMatrix matrix)
{
   int      numDimensions, dimIndex;
   double   av_distance,
            *centroid,
            *dummy;
   cluster  *dummyCluster, *clustIndex;
   vector   *dummyVect;

   numDimensions = matrix -> dimensions;

   if (!matrix -> centroid)
      matrix -> centroid = ClusterCentroid(matrix -> vector_set);

   if (matrix -> num_vectors == 1) {
      av_distance = INITIAL_DISTANCE;
   } else {
      clustIndex = matrix -> vector_set;
      av_distance = 0.0;
      while (clustIndex != NULL) {
         av_distance += VectorEuclideanDist(clustIndex -> vect, matrix -> centroid);
         clustIndex = clustIndex -> next;
      }
      av_distance /= (double) matrix -> num_vectors;
   }

   dummyCluster = ClusterNew();
   centroid = matrix -> centroid -> components;
   dummy = (double *) malloc (numDimensions * sizeof(double));
   dimIndex = 0;
   while (dimIndex < numDimensions) {
      memcpy(dummy, centroid, numDimensions * sizeof(double));
      dummy[dimIndex] += av_distance;                        /* +ve along dimension */
      dummyVect = VectorConstruct(numDimensions, dummy);
      dummyCluster = ClusterAddVector(dummyCluster, VectorMakeCopy(dummyVect));
      VectorDeallocate(dummyVect);

      memcpy(dummy, centroid, numDimensions * sizeof(double));
      dummy[dimIndex] -= av_distance;                        /* -ve along dimension */
      dummyVect = VectorConstruct(numDimensions, dummy);
      dummyCluster = ClusterAddVector(dummyCluster, VectorMakeCopy(dummyVect));
      VectorDeallocate(dummyVect);

      dimIndex++;
   }

   free(dummy);
   return dummyCluster;
```

```
}


void CalculateAverage(corrMatrix matrix)
{
   double   correlation, sum, sumSqr;
   int      numVects;
   cluster  *dummyCluster, *cIndex;

   if (CMatrixUseDummy(matrix)) {
      cIndex = dummyCluster = MakeDummyCluster(matrix);
   } else {
      cIndex = matrix -> vector_set;
   }

   sum = 0.0;
   sumSqr = 0.0;
   numVects = 0;
   while (cIndex != NULL) {
      correlation =  CMatrixCorrelationWithVect(matrix, cIndex -> vect);
      sum += correlation;
      sumSqr += SQR(correlation);
      cIndex = cIndex -> next;
      numVects++;
   }

   if (CMatrixUseDummy(matrix)) {
      ClusterDestroy(dummyCluster);
   }

   matrix -> avCorrelation = sum / (double) numVects;
   matrix -> stddevCorr = ((double) numVects * sumSqr) - SQR(sum);
   matrix -> stddevCorr /= (double) SQR(numVects);
   matrix -> stddevCorr = sqrt(matrix -> stddevCorr);

   matrix -> validAverage = TRUE;
}

void WriteHex(char buffer[], const unsigned char *data, size_t numBytes)
{
   size_t   byteIndex;
   char     hex[5];

   buffer[0] = 0x0;        /* Buffer initially null */
   for (byteIndex = 0; byteIndex < numBytes; byteIndex++) {
      if (data[byteIndex] < 0x10)
         sprintf(hex, "0%x", data[byteIndex]);
      else
         sprintf(hex, "%x", data[byteIndex]);   /* Get the hex character */
      strcat(buffer, hex);
   }
}

void ReadHex(const char buffer[], unsigned char *data, size_t numBytes)
{
   size_t      byteIndex;
   char        hex[5];
   short int   dec;

   for (byteIndex = 0; byteIndex < numBytes; byteIndex++) {
      hex[0] = '0';
      hex[1] = 'x';
      hex[2] = buffer[byteIndex * 2];
      hex[3] = buffer[byteIndex * 2 + 1];
      hex[4] = 0x0;

      sscanf(hex, "%x", &dec);
      data[byteIndex] = (unsigned char) dec;
   }
}


/* --- END of corr_matrix.c --- */
```

Technical report last page