HARVEY MUDD COLLEGE

Computer Science Clinic

Final Report for
Sandia National Laboratories

Implementation of and Experimenting with a Clustering Tool

February 21, 2006

Team Members
Avani Gadani (Team Leader)
Daniel Lowd
Brian Roney
Eric Wu

Advisor
Prof. Belinda Thom

Liaison
Dr. Kevin Boyack

Abstract

The goal of the 2002-2003 Sandia National Laboratories Computer Science Clinic Project was to create a tool for simultaneous visualization of several different reductions of multi-dimensional data sets and their analysis. Analysis was done by implementing manual clustering and several automatic clustering algorithms including k-means, linkages, and DBSCAN density. Validity metrics were implemented to quantitatively compare different clusterings of the same data, assess the relative fitness of a clustering, and determine the likelihood that a clustering is different from a random one. The tool was given to Sandia and was used to analyze data sets ranging from journal cross-citations to genomics data.

Contents

List of Figures

Chapter 1

Background

1.1 Problem Statement

Sandia National Laboratories in Albuquerque initially requested a tool that would enable them to view multiple two-dimensional plots of data sets simultaneously and to correlate points across the plots. The correlation would be done by letting the user interactively select a set of points in one plot, and then highlighting the points with the same unique identifiers in every other plot.

The problem was then extended to include clustering algorithms and validity metrics along with experimentation using the tool on both contrived and actual data sets in hopes of learning more about both the data and the tool.

1.2 The Clustering Problem

The project scope implied that Sandia wanted to be able to interact with and analyze their data, so the team designed the tool to support both clustering by hand, i.e., assigning every point a label manually, and clustering by machine, where the points are clustered automatically using various algorithms.

Clustering is the process of taking an n-dimensional data set and partitioning it into groups of points where the groups are selected based on proximity or some other similarity metric. On this project, the data sets that the tool had to deal with were all two-dimensional and thus the clustering algorithms were looking at the geometric properties of the data sets.

While some data sets have closely knit groups of points that are scattered throughout the plot, most data sets exhibit some ambiguity as to what is a "correct" clustering. For example, in Figure 1.1 it is difficult to judge whether the bulges on the side of the plot are meant to be clusters or not. If one algorithm places those points in clusters and another does not, some sort of quantitative metric may assist in determining which clustering fits the data best. However, most such metrics are subjective and domain specific, so a given metric may or may not work on a given algorithm. The final answer can only be found through human interpretation based on the similarities in the original data set, which makes solving the problem computationally intractable.

There is no conclusive answer to the clustering problem, as even two humans are likely to have a different idea of what exactly a "good" clustering is.

Figure 1.2 displays two clusterings of a particular data set, each done manually by a human being. They are both "ideal" clusterings in the sense that they have been clustered exactly as a human would perform the task, yet they are still not identical.

Validity metrics are one method for quantitatively determining the similarity of a pair of clusterings. Ideally, given many different two-dimensional reductions of a multi-dimensional data set with different similarity computations, validity metrics would help automatically suggest similarities and ordinations that are worthy of further study, as well as the algorithms most suited for clustering the data set.

1.3 Data Analysis at Sandia

Sandia's current approach to analyzing data begins with a data set with some arbitrary number of features. Each data point carries a unique ID and values for various features defining the data. The features depend of the type of data being studied, but may include data such as abstracts, citations, and keywords in a title for a data set consisting of technical articles, or pitch and tempo of a song over time in the case of a data set of musical pieces.

In this form, a data set is not amenable to either automatic clustering, or to human insight. The dimensionality of the data (i.e., the number of features) is likely to be too high to be easily visualized by a person, and the features are too difficult for a computer to quickly determine.

Defining distance metrics between features is necessary for clustering to take place. Sandia deals with the problem by defining several different similarity metrics. These metrics may be as simple as Euclidean distance between numerical data, or as complex as necessary, e.g., number of papers cited in common between two articles. These similarity metrics produce a set of pairwise distances in a .sim file. Each row in a .sim file consists of two unique ID's and the pairwise similarity between them. Note that not all possible pairs of data points may be represented, due to incomplete information in the original data set or lack of similarity.

The various .sim files can then be sent to VxOrd, a graph layout program. The similarity metric in conjunction with VxOrd acts as a dimensional reduction of the original data to a 2-dimensional plane. Based on various parameters, VxOrd stochastically converts the .sim file to a 2-D scattergraph, with each point representing one of the unique ID's in the original data set, and distances between points roughly corresponding to the inverse of the similarity between them. A given layout of points on the x-y plane is referred to as an ordination. Different values for VxOrd's parameters will result in different scattergraphs with various levels of clumping. An ordination provided by VxOrd can be output as a .coord file, consisting of each unique ID and its x- and y-coordinates.

The .coord files can either be viewed via VxInsight, a visualization tool discussed later, or sent to our MATLAB clustering tool. The tool can run various clustering algorithms to cluster the ordination from VxOrd. These automatic clustering algorithms will be explained in further detail later. The tool can also open multiple .coord files based on different similarity measures or different ordinations based on the same similarity measure side-by-side. With multiple ordinations open, a user can choose points in one plot either manually or from an automatic clustering and view where the corresponding points in the other plots are placed. This functionality allows a user to identify points in other plots that appear together as a single cluster in a particular ordination. This is useful for human validation, since if a cluster tends to remain together in ordinations based on keywords in titles and co-citations, for example, then it can be inferred that articles with similar keywords in titles are likely to cite the same papers. The tool also supports several automatic validity measures, both external (comparing the clusterings of two plots to each other) and internal (measuring the clustering of a single plot). These will also be explained in further detail later.

As shown in Figure 1.3, once the task of reducing data to a easily visualized state is completed, and one of many ordinations is chosen for further study, VxInsight can be used to manually explore the data. VxInsight is a visualization tool that links the .coord files to the original data set (in the form of a Microsoft Access database) and shows a 3-d plot of the ordination. At this point(Figure 1.4), human investigation of clusters in detail can begin.

1.4 Project Goals

The goals set forth at the beginning of the year were:

1. To allow a user to visually compare different coordinate files produced using different features or random seeds.
2. To provide functionality for clustering the data using several algorithms, and to automatically compare clusterings using known validation methods.
3. To ultimately allow the user and the clustering algorithm to work together to produce a better clustering solution than either could individually.
4. To allow a user to experiment with different validity metrics over clusterings.

The first two goals have been met completely. The third point is not so much a goal as a light over the horizon. Writing an optimal clustering program is equivalent to solving the hard AI problem, which is beyond the scope of this project.

Validity metrics, the fourth goal, have also already been implemented, but there is still some work to be done to make the validity measures a selling point of the product.

1.4.1 Labor Management

Our timeline for the year was as follows:

Sept. 23rd	Project Proposal.
Oct. 17th	Deadline for preliminary basic cluster-viewing tool.

Oct. 18th	Site Visit, Sandia National Laboratories: Albuquerque, NM.
Oct. 29th	Presentation: Initial Peer Review of Project.
Nov. 05th	Finish basic cluster-viewing tool.
Nov. 17th	Deadline for report on existing clustering metrics.
Nov. 17th	Implement at least one clustering algorithm.
Nov. 24th	Decide on and implement at least one metric for one clustering algorithm. Implement 2nd clustering algorithm if we have time.
Dec. 5th	Draft of Mid-year report.
Dec. 10th	Presentation.
Dec. 12th	Mid-year report.
Late Dec - Early January	Implement nearest-neighbor validity.
January	More clustering metrics (e.g. Gaussians).
January	Make necessary GUI changes.
February-March	Do more experiments and begin work on a paper.
Apr. 22nd	Code Freeze.
Apr. 25nd	Poster Due.
May 9th	Final Report/CD.

These deadlines were, for the most part, met or superseded. We had the good fortune to discover K-means and five variants of linkage algorithms already implemented in MATLAB, which sped up our progress considerably.

Over the second semester, our focus was supposed to switch towards research. Unfortunately, the team member assigned to explore this option was unable to obtain results positive enough to justify switching attention away from code and feature optimization in the tool. We instead implemented several validity metrics and an additional clustering algorithm [1].

The tool was also sent to Sandia several times during this period and all requested revisions were made.

[1]The seven metrics did not include nearest neighbor, which we dropped at the suggestion of our liaison.

1.4.2 Deliverables

- Clustering and Visualization Tool (MATLAB)
- Tool Documentation
- Final Report
- CD with all of this and all papers, presentations, and papers referenced for this project.

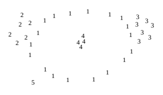
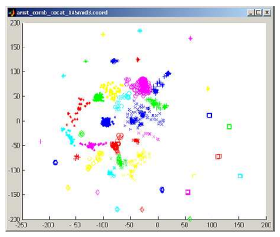
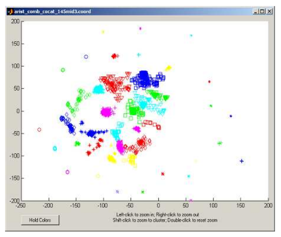


Figure 1.1: A possible clustering



(a) Human A



(b) Human B

Figure 1.2: A comparison of hand clusterings of the same data set done by different people.

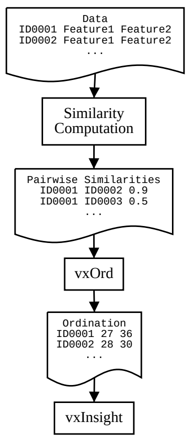


Figure 1.3: Sandia's current process. Starting with a database of identifiers and information about them (features), a pairwise similarity file (.sim file) is generated using some similarity computation algorithm operating on a subset of the features. Using VxOrd, each identifier is then mapped to an x-y coordinate so that the Cartesian distance between two points roughly reflects their similarity. Finally, VxInsight is used to visualize the data.

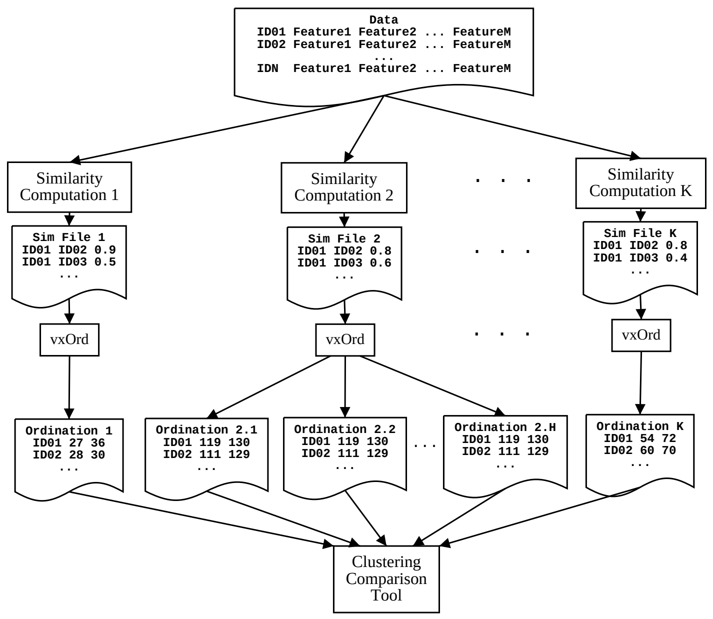


Figure 1.4: Extension to process, for comparing different similarity metrics. The tool we developed can read multiple ordinations generated from different similarity files, which in turn are generated using different similarity measures.

Chapter 2

Clustering Algorithms

2.1 K-means

K-means clustering is a commonly used partitional algorithm (Jain et al., 1999a). The standard K-means clustering requires choosing a number of clusters, k, and assigns each data point a label corresponding with its cluster membership. The algorithm works as follows: First, choose k cluster centers. Determine each data point's cluster membership according to the cluster center closest to it by a distance, d. Recompute the cluster centers based on the actual cluster membership. Then repeat until some criteria is met, such as a lack of data points being reassigned to new clusters. The underlying k-means process minimizes

$$E = \sum_{i=1}^{c} \sum_{x \in C_i} d(x, m_i) \tag{2.1}$$

, where m_i is the center of cluster C_i.

The run-time for the K-means algorithm is $O(k \circ n)$ for each iteration. However, the number of iterations necessary is unknown since standard k-means is not guaranteed to converge. In addition, clusterings produced by k-means are dependent on the starting points of the clusters. Variants of k-means exist which are guaranteed to produce an optimal clustering from any starting points (**?**). The advantages of this algorithm are that it runs in time linear to the cardinality of the data set, and modifications to the algorithm can be made to guarantee an optimal partitioning (Jain et al., 1999a). An inherent issue in k-means clustering is that convex clusters tend to be produced when Euclidean distance is used as the distance measure.

2.2 Single Link

Single-link clustering can broaden one's search to include non-convex clusters, though the algorithm introduces other issues (Halkidi et al., 2001a). Single-link is an agglomerative hierarchical algorithm, which assigns each data point to an individual cluster, then merges clusters together according to the minimum of all pairwise distances between clusters members. The advantage of merging clusters based on distances between distinct members of each cluster is that non-convex clusters may be formed, such as clusters consisting of concentric circles.

However, this tendency may also produce clusters that are "chained", or non-compact. In addition, the process of determining the minimum of distances is relatively computational expensive; the algorithm runs in O($n^2 \log n$).

2.2.1 Other Linkages

Complete-link Complete-link measures the distance between clusters as the maximum distance between two points within different clusters. This favors tight and compact clusters.

Average-link Average link uses the average distance between two points within different clusters. This is a compromise between the risk of error in single-link and complete-link's propensity for extremely tight clusters.

Centroid-link Centroid-link uses the center of mass of each cluster to calculate distance. Note that this has the possibility of having some problems - two clusters may merge in such a way as to make the resulting distance to some third cluster to be less than either original cluster was to the third cluster before the merge. If this occurs, MATLAB will print a warning, and suggests that a different method be used.

Ward-linkage Ward link uses the "inner squared distance", or the increase in the sum of squares of the distance between all points in either cluster caused by merging the two groups. In simpler terms, if clusters A and B are to be compared, and have a and b numbers of points respectively, the ward distance is $\frac{ab}{(a+b)\times Centroid^2}$, where $Centroid$ is the distance as calculated by centroid-link above.

2.3 Density

Density clustering using Sander's DBSCAN algorithm is based on the principle that when no information is available for a data set besides the 2-d plot, a simple visual inspection of the plot may suffice to find clusters (Ester et al., 2000). It has been suggested that an intuitive way to cluster a plot is to find groups of points that are both close to each other and also farther away from other groups of points and assign them the same labeling. Density clustering attempts to do exactly such a thing. The user may look at a plot and visually determine an estimate for the density of the plot. The input parameters for the algorithm are the radius of the average cluster (`EPS`) and the minimum number of points (`minPoints`) within the radius needed to recognize a relatively dense area of the plot. The DBSCAN algorithm steps through each point identifying those pairs of points which symmetrically satisfy the density criteria and giving those points the same label.

Points that are not within an `EPS` radius of any other non-noise point or do not have `minPoints` points in their EPS radius are treated as noise. All noise points are given the same label by our tool, in which they are represented by black dots. While the actual DBSCAN algorithm stores the data groups in R* Trees to achieve O(n) run-time, our current implementation does not, and runs in O(n^2).

Chapter 3

Cluster Validity

3.1 Overview

One of the primary goals of this project was to simplify the task of clustering data sets similarly to how a human might do it. With that in mind, we have implemented various validity metrics to evaluate the quality of an automatic clustering. In addition, since all of our clustering algorithms are dependent on the geometry of a data set, comparative measures have been implemented to quantitatively define the difference between an automatic clustering and a possible clustering known *a priori* based on the underlying data.

3.2 Internal measures

Internal criteria measure how well a clustering fits the geometric structure of a data set with no reference to information known *a priori*. By using Monte Carlo testing, the quality of a clustering can be measured without falling back on the recourse of comparing to a human clustering, which of course is not always available.

3.2.1 Davies-Bouldin

The Davies-Bouldin index is a measure of the uniqueness of clusters in a given clustering. A measure for dispersion of a single cluster and dissimilarity between a pair of clusters is required. A simple definition for each is that the dispersion of a cluster S is the average Euclidean distance of all points to the cluster center. The dissimilarity D is defined as the distance between the two cluster centers.

From these measures, a similarity metric, R can be defined for pairs of clusters. The metric must satisfy the following conditions:

1. $R_{ij} \geq 0$
2. $R_{ij} = R_{ji}$
3. If $S_i = 0, Sj = 0$, then $R_{ij} = 0$
4. If $S_j > S_k$, and $D_{ij} = D_{ik}$, then $R_{ij} > R_{ik}$
5. If $S_i = S_k$ and $D_{ij} < D_{ik}$, then $R_{ij} > R_{ik}$

The first two conditions state that R is non-negative and symmetric. The third states that two individual points are always entirely unique. The fourth asserts that all other things being equal, a well-dispersed cluster is more similar to all other clusters than a tight cluster. The fifth simply states that closer clusters are more similar.

The choice for R implemented in our tool is:

$$R_{ij} = \frac{S_i + S_j}{D_{ij}}$$

Other variants of the Davies-Bouldin index using minimum spanning trees, relative neighborhood graphs, and Gabriel graphs are discussed in (Halkidi et al., 2001b).

Given a similarity metric, we then define the mundanity, M, of a cluster in a plot with nc clusters as:

$$M_i = \max_{j=1:nc, j \neq i} R_{ij}$$

Since very similar clusters will have high values for M, clearly a low M value represents a unique, compact, and well-separated cluster.

The Davies-Bouldin index is then simply defined as:

$$DB = \frac{1}{nc} \sum_{i=1:nc} M_i$$

The Davies-Bouldin index then becomes the average similarity of each cluster with its most similar cluster. Since we are searching for unique rather than similar clusters, a minimal value for this index represents a good clustering. Since the index is an average of the mundanity of each cluster, this index is invariant with respect to number of clusters.

3.2.2 Hubert's Γ statistic

Hubert's Γ statistic measures the correlation between two square matrices of the same size. The modified Hubert's Γ statistic uses the proximity matrix of all the points in the data set as the first matrix and the proximity matrix of the centers of the cluster to which each point belongs as the second matrix. For a data set with N points and $M = \frac{N(N-1)}{2}$ pairwise comparisons of points, we define P as the $N * N$ matrix such that $P(i,j)$ is the proximity of point i to point j. Q is a matrix of the same size where $Q(i,j)$ is the proximity of the representative member of the cluster containing i to the representative member of the cluster containing j. In our implementation of Hubert's Γ statistic, the representative member of a cluster is the center of the cluster. Then,

$$\Gamma = \frac{1}{M} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} P(i,j) \times Q(i,j)$$

A high value of Γ implies well-separated clusters. However, as the number of clusters increases, Γ also increases. Therefore, there is also a normalized version of Hubert's Γ statistic:

$$\hat{\Gamma} = \frac{\frac{1}{M} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} (P(i,j) - \mu_P) \times (Q(i,j) - \mu_Q)}{\sigma_P \times \sigma_Q}$$

μ is the mean and σ the variance of all the elements in the matrix. $\hat{\Gamma}$ is scaled from -1 to 1, and a large absolute value of $\hat{\Gamma}$ implies well-separated and compact clusters.

3.2.3 Monte Carlo testing

Monte Carlo testing is used to determine just how valid a clustering is for a given data set. The internal validity metrics supply a numerical value, but that number is meaningless without knowledge of the expected distribution of values for that metric. Computing the actual expected distribution for a large data set is incredibly difficult, however. In lieu of calculating a precise distribution, Monte Carlo testing is used to approximate the distribution.

In more precise terms, we use the random label hypothesis, H_0, also known as a null hypothesis. H_0 states that all permutations of labels on a set of objects are equally likely (Theodoridis and Koutroumbas, 2003). We hope to use Monte Carlo testing to disprove H_0, thus showing that a given clustering is unlikely to be random.

The probability density function (pdf) of a metric is computed by generating many random sets of labels for a data set. For each set of labels, the metric is run and the values stored. The histogram of values is an approximation of the pdf.

Using the approximation of the pdf, comparison of the calculated value v of the metric on the non-random clustering to the pdf reveals the likelihood that H_0 is true. Assuming that a metric has a right-tailed distribution, if v is less than a significant portion of the pdf, then H_0 can be rejected, meaning that the clustering is unlikely to be random, or that the clustering matches the structure of the data set well. The divergence between v and the pdf is indicative of the confidence with which H_0 can be rejected. Of course, if v falls within a common area in the pdf, then we accept H_0 and conclude that the clustering in question does not fit the data set. For left- or two-tailed distributions, values of v greater than the pdf or otherwise significantly different from values expected from the pdf indicate a non-random clustering.

3.3 External measures

An external measure is used to quantify the similarity between one clustering and another. Our tool has implemented several measures based on classifying how pairs of points relate in each clustering.

Each pair of points is classified as SS (Same-Same), SD (Same-Different), DS (Different-Same), or DD (Different-Different) depending on what clusters the points belong to in each clustering. For example, consider two clusterings C_1 and C_2. If two points are in the same cluster in C_1 and in the same cluster in C_2, the pair is classified as SS. If they are in the same cluster in C_1 but not in C_2, then they are classified SD. In a similar vein, DS and DD are defined. From classifying each pair of points, using $M = SS + SD + DS + SS$, $m_1 = SS + SD$, and $m_2 = DS + DD$, the following measures can be defined:

- Rand statistic:
$$R = \frac{SS + DD}{M}$$
- Jaccard coefficient:
$$J = \frac{SS}{SS + SD + DS}$$

- Fowlkes and Mallows index:

$$FM = \sqrt{\frac{SS}{SS + SD} \times \frac{SS}{SS + DS}}$$

- Hubert's Γ statistic:

$$\hat{\Gamma} = \frac{M \times SS - m_1 m_2}{\sqrt{m_1 m_2 (M - m_1)(M - m_2)}}$$

The first three measures are scaled from 0 to 1, with larger values indicating a high degree of correlation between the two clusterings. The last measure is scaled from -1 to 1, with large absolute values meaning the two clusterings are similar. Note that the maximum values are attainable only when $nc_{C1} = nc_{C2}$, although the measures may still be calculated when $nc_{C1} \neq nc_{C2}$.

In practice, R degrades as a useful measure as nc increases, due to the DD term overwhelming all other terms. For example, in a data set with a large number of points and clusters, any randomly chosen two points are likely to be in different clusters in both C_1 and C_2. J attempts to adjust for this issue by ignoring the DD term. FM further adjusts by making a distinction between the SD and DS terms. The logical underpinning of FM is that it looks at all pairs within C_1 and all pairs in C_2 and takes the geometric mean. These terms may vary widely in the case of two clusterings with widely varying distributions of points within clusters. For example, if C_1 was generated using K-means, each cluster is likely to contain approximately the same number of points. If C_2 has been clustered using a different algorithm, such as single-link or density clustering, then there may be a wide variation in the number of points per cluster. If these two clusterings are compared, the DS term is likely to rise, as many points in C_1 that are in different clusters will be lumped into the same cluster in C_2 if there are particularly dense areas of the data set that have been mistakenly partitioned into separate clusters by K-means. When comparing two clusterings with a uniform number of points per cluster, however, $FM \approx \sqrt{J}$.

$\hat{\Gamma}$ in this context is a simplification of the same statistic discussed above in the internal measures section. In this case, the matrices being compared are the membership matrices M for each clustering. M is defined as follows:

$$M(i,j) = \begin{cases} 1 & \text{if i and j are in different clusters} \\ 0 & \text{otherwise} \end{cases}$$

Using the simplifying assumption that $SD \approx DS$, the formula for $\hat{\Gamma}$ can be reduced to

$$\frac{(SS \times DD) - SD^2}{(SS \times DD) + SD^2 + (SS \times SD) + (SD \times DD)}$$

Looking at what happens if any one term becomes dominant (as DD tends to become with large nc, or SD in dissimilar clusterings), we can see that a large SD will reduce the equation above to $\frac{-SD}{SD+SS+DD}$, tending towards -1. A large $DD \times SS$ term will similarly tend towards 1. Note that $SS \times DD$ maximizes when $SS = DD$ (assuming a strict give and take between the two), and assuming a large DD, as is common among data sets with large nc, this will correspond to SS being large compared to SD.

3.4 Relative measures

There will be occasions where the question is not of whether a particular clustering is good, but rather which from a set of possible clusterings is best. One such example would be in attempting to find the number of clusters nc in a data set. Relative measures assist in choosing which value of nc is best according to a prespecified criterion, usually an internal measure.

If the internal measure being used is not proportional to nc, then finding the best clustering among a set is easy. Simply finding the minimum (in the case of Davies-Bouldin) or the maximum (normalized Hubert Γ statistic) in a plot of index values vs. parameter values will show which parameters result in the best clustering.

If the measure used increases as nc increases, such as in the case of the modified Hubert Γ statistic, then simply finding the minimum or maximum on a plot is no longer sufficient. Instead, a significant local change in the value of the measure, seen as a "knee" in the plot, is indicative of the best parameters for clustering. The absence of such a knee might be an indication that the data set possesses no clustering structure. (Theodoridis and Koutroumbas, 2003) Due to the difficulty in objectively ascertaining the existence of a knee in a plot, the normalized statistics are preferred for use in relative measures.

Chapter 4

Exploratory Research

4.1 K-means repeatability

The K-means algorithm is by its nature non-deterministic. While this is expected, it was still surprising how different multiple runs of K-means on the same data set could be. There were some unhappy surprises. For instance, while running K-means on a real data set that appeared to have a fairly obvious partition into fourteen clusters, we did not receive as good of a clustering using the K-means algorithm as one might have expected.

As such, we felt it worth investigating just how repeatable K-means was on certain data sets. It was discovered that in some cases K-means is repeatable, in particular on the 34-point “Karate” data set, a data set with two circular groups with some overlap.

However, in a data set with more separation between clusters but more variation in cluster size, the clusterings were often completely different, receiving a Jaccard coefficient of 0.5, indicating that the Same-Different and Different-Same pairings were just as prominent as the Same-Same pairings. There was some thought that this might be dependent on number of clusters, and the “best fit” number of clusters would have a spike in the Jaccard measure (with the idea that K-means could be more stable with a number of clusters fitting the graph best), but none was visible. Instead, there was a flat line, indicating that the Jaccard coefficient was approximately 0.5 regardless of number of clusters. A similar run on a “real” data set returned similar results.

Discussion and investigation led to the realization that this happened because the data sets examined had varying cluster sizes - one cluster would have ten points in it, while another would have fifty or so. K-means at-

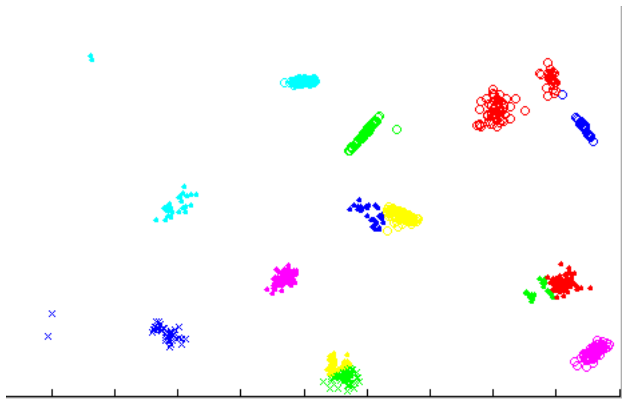


Figure 4.1: A poor clustering- notice the dense groups of points with more than one color present.

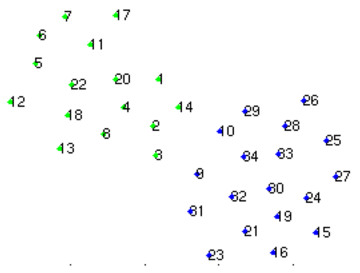


Figure 4.2: This is a very consistent k-means coloring. Each run will return the same labelling.

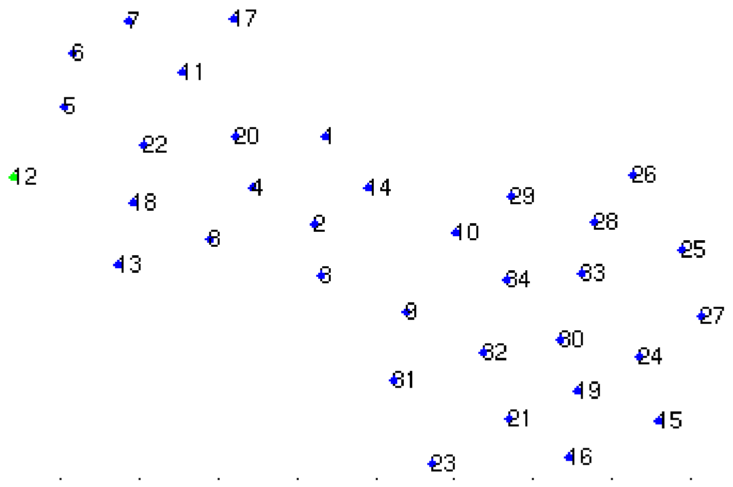


Figure 4.3: A very bad single-link clustering. The point on the left (number 12) is just a little too far away from its neighbors.

tempts to minimize the sum distance of the points to their cluster center. Occasionally, having two cluster centers in the size fifty cluster is "correct" according to K-means' optimization criteria. This behavior is what was observed. Our conclusion is that K-means is not the appropriate clustering algorithm when the natural structure of a data set requires a solution with very different cluster sizes. This result corresponds with results found researching the issue (Duda and Hart, 1973).

4.2 Single-link

During a run of Jaccard/Fowlkes comparisons, one data set/clustering algorithm stood out: the "Karate" data set, which consists of 34 data points in two slightly overlapping, roughly circular groups, single-link clustering separated the graph into a single point cluster on the left side, and everything else in another cluster on the right. All of the other linkage algorithms, k-means, and density clustering returned quite reasonable values.

Single-link computes an answer based on finding the minimum paired distance between a point and its neighbors. What happened in this case is that a single point on the left had the distance between it and its neighbors slightly higher than the points at the transition between the overlapping

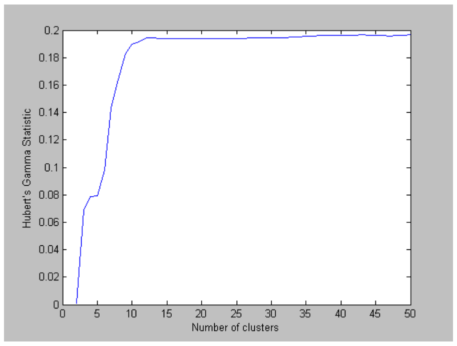


Figure 4.4: An example of a Hubert Γ graph with no obvious "knee". It is unclear what the preferred value for the number of clusters would be.

sections. Single-link, as it evaluates, saw the smaller, overlapping section and proclaimed it part of a chain, while the individual point was further away and proclaimed separate. We suggest that in the future *data sets be eyeballed to determine which clustering algorithm is best*, as we have experienced data sets that break individual clustering algorithms to the point of uselessness. In single-link's case, the telltale sign is a very close chain connecting different groups, possibly closer than points within the clusters themselves.

4.3 Hubert's Γ statistic

A run of Hubert's Γ statistic on a very clear set of 14 clusters using single-link returned unclear results. In this particular example (see graph), there was no particularly obvious knee in the graph.

A run of Hubert's Γ statistic on a very clear set of 14 clusters using single-link returned unclear results. In this particular example, there was no particularly obvious knee in the graph. This is likely (although not definitely) because the last few clusters to be "discovered" were very small - two or three points in an outlying group. Normalized Hubert Γ, however, showed a maximum at 14 clusters, which, given the structure of the data set, is a reasonable value. Independent functions provided by the user can easily generalize this technique to other graphs - the clustering (linkage) of a plot only needs to be done once, after which the cluster division is relatively fast. Due to the consistency of the linkage algorithms, it can be said with confidence that the resulting graph of Hubert Γ values reflects the

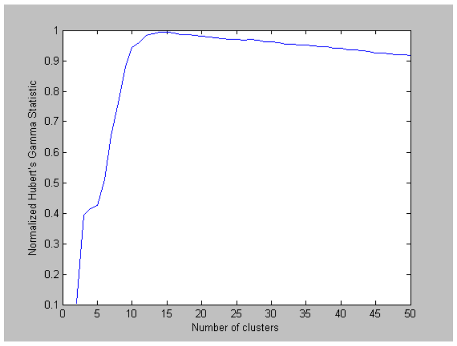


Figure 4.5: The Normalized version of Hubert Γ. It's hard to see from the graph, but there is a slight maximum at 14, the preferred number of clusters.

"optimal" number of clusters using that algorithm. K-means, if used in this manner, should, due to the randomness inherent in the initial conditions, be run several times and an average normalized Hubert Gamma used.

4.4 Comparison against random

There has been some discussion as to the usefulness of the Rand / Jaccard / Fowlkes-Mallows measures when used against a random data set. On some plots, the value of these measures when comparing K-means or single link or the others against random is the same as if someone were to compare a different random coloring to the original random coloring. In others, K-means scores might be slightly higher. It was not clear what causes this variation, as occasionally those clusters that looked "better" were not the ones that scored higher scores when tested against random. Observe that the Rand/Jaccard/Fowlkes measures do not keep track of the structure of the colorings. No distances or positions are remembered, just the relative colorings are considered. Given this, we theorized that Rand/Jaccard/Fowlkes comparisons when run against random clusterings are very uninformative. For example, K-means usually scores against a random coloring about as well as two random colorings. This was found to not be the case when a single cluster has the majority of the points in a graph, and we believe that this is due to the fact that what the K-means or single-link compared to a random Jaccard test really measures nothing more than the relative sizes of the clusters within the graph. A higher variation among cluster sizes will raise the likelihood of SS matchings, while decreasing DS

matchings. It should be noted that K-means will not do well in outscoring random on this test compared to the other clustering algorithms, because it attempts to minimize the sum of the distance from the center of the cluster to the points within that cluster. What this really means is that K-means is biased to split a "dense" grouping of points into several cluster centers (rather than just one) so as to minimize the total sum, and this additional division will lower the comparison against random accordingly. This is not meant to belittle the usefullness of the value of the Jaccard testing against random. It provides a quite useful base - if a graph being compared to K-means scores worse on this metric than random does, that graph should be reexamined. But one should not consider any validity measure blindly, just as one should not rely on a clustering algorithm's solution without considering the big picture.

4.5 Practical uses

We decided that it would be useful to make a case study of the tool, and so we asked the Auditude clinic team to provide some of the data they were exploring (they had previously expressed interest in our tool). When their data, obtained from audio files of several thousand songs, was plotted in our tool, it looked like just a big, circular blob, in which no structure was to be seen (the data provided was already two dimensional coordinates, so we did not send the data through VxOrd, which would have probably provided a more useful plot if the original data had more information). However, despite the lack of visible structure, using our tool's ability to load colors from a separate graph, a plot was created that consisted of just a particular artist, which, when colored, showed up on the original graph. Several of these supported the hypothesis that there was a structure based on instruments involved in the songs, which is exactly what they were hoping for. From a mess of no relative information, being able to highlight points in a separate file very clearly showed which area of the graph was classical, which area was a cappella, etc. This showed the Auditude clinic team that the technique they were investigating was in fact useful, and saved them much time in the testing phase.

Chapter 5

Conclusion

The tool our clinic created was greatly useful to ourselves in our investigative experiments, as well as for other clinic teams that were strugging with the need to cluster complex data. The ability to see the effects of one graph's coloring on another, the ability to see how different clustering algorithms affect certain validity measures, and how points can migrate or stay together across different similarity metrics is very powerful, and now, with our tool, it is available for use.

Our clinic team considers this clinic project to be a success. We have created a tool that satisfies the original requirements, and proceeded to learn about and implement several additional features. Weaknesses and strengths of various automated algorithms have been discovered and catalogued, and independent groups have evaluated and appreciated the tool.

Clustering is one of the hardest, most general artificial intelligence problems that is actively being pursued in a wide variety of fields. A consistent, accurate algorithm to divide data into groups and classifications with no human aid would be a universal boost in capability. Unfortunately, the current state of technology does not include such an algorithm. Instead, we are required to guide algorithms and verify their results, making sure that mistakes are not made, and manually labelling the points when needed. This is certainly not optimal. Our tool offers a new interface to blending both human input and control with automatic and semi-automatic algorithms. The work available to us in the field has been gathered to provide multiple clustering and validity algorithms to fit the need of the user.

Appendix A

README

```
README
Sandia Clustering Tool, release 1.0
5/9/2003

TABLE OF CONTENTS

1. Quickstart
2. Using the Tool
3. Scripts for validity testing
4. Known Issues
5. List of Included Files


1. QUICKSTART

To install the tool, simply copy all the files to a
separate directory.  This program requires MATLAB R13
or more recent, including the stats package.

To run, change to the install directory using the CD
command.  To run the tool, just type ’controller’.


2. USING THE TOOL

The first window to appear will be the Controller,
```

containing nine buttons: Open, Cluster, Select, Clear Colors, Save Colors, Load Colors, Validity, Close Plots, and Tile.

The Open button enables the user to select an ordination for display in a new plot window. The required file format is point name followed by x and y coordinates, the same used by vxOrd for saving .coord files.

Newly opened plots display all points from the original .coord file as black dots. The way points or groups of points are displayed in a plot may be changed by automatic clustering (the Cluster button), manual clustering (the Select button), or loading a saved clustering (the Load Colors button). By default, these commands simultaneously modify how points in all plots are displayed. To avoid this behav-ior,
each plot window has a toggle button labelled "Hold Colors". When selected, the plot’s clustering is "frozen" so that it remains unmodified as other plots change. The one exception to this is manual clustering: a frozen plot may be manually clustered, but these modifications will not be applied to the other plots.

The Cluster button opens a Cluster dialog box. From this box, the user may specify a plot to be clustered, an algorithm to use, and additional parameters. The number of clusters parameter applies to all algorithms except density; the EPS parameter, however, only applies to the density algorithm. The EPS parameter is meant to be an estimate of the radius of any cluster in the graph. It is suggested that the user load the plot, examine the largest desired cluster group, and choose an EPS value appropriate to that graph. Be warned that clustering large datasets may take a long time. When the clustering is finished, it will color all open and unfrozen plots according to the clustering’s labelling. Points assigned no label will be shown as black dots; the controller will never color discovered clusters black.

The Select button opens the Select Points dialog box, and puts all plots into select mode. In select mode, clicking and dragging a box on any plot will recolor those points to be the current select color and symbol. If the plot is unfrozen, all corresponding points in other unfrozen plots will be recolored as well. The Select Points dialog allows the user to select the color and symbol to use for this. Select mode is turned off when the Select Points dialog is closed.

When not in select mode, all plots are in zoom mode by default. Zoom works mostly as with other MATLAB programs: left-click zooms in, right-click zooms out, double-click zooms all the way out. Though plots are colored together, they can be zoomed independently. The one additional zoom feature for this tool is ’zoom to cluster’. A shift-left-click will zoom *all* plots to show all points of the nearest cluster. Experimentation will clarify this feature better than additional explanation.

The Clear Colors button resets the coloring of all points in all unfrozen plots to be black dots.

The Save Colors button opens a dialog enabling the user to save the coloring of one plot to disk. A plot may be selected from among those open, then by clicking "Save as..." the desired file name may be selected. The file format for saved files is: <id> <colorindex>, where colorindex is an integer that specifies the color and symbol to be used for the datapoint. A .color extension is automatically appended to the file name. This button is only active when at least one plot is loaded.

The Load Colors button allows the user to load a coloring file from disk. Once loaded, all currently open and unfrozen plots will be recolored to match the specified coloring. IDs whose colors are not specified by the file are colored as black points. The file format for coloring files matches that used by the Save Colors action.

The Validity button opens a dialog enabling the user to run various validity metrics on currently loaded plots or their clusterings. The general validity metric, specific type of test to run (not available for all metrics), and plot or plots to use may be selected from popup menus. The "Run Test" button will run the currently specified test and display the results below in the Results edit box. Some tests may take a while to run, depending on the number of points in the plots. For convenience, both the description of the experiment and its results are displayed in the Results box and may be copied and pasted to an editor for saving. The Rand-type statistics can be called with either one or two plots as arguments. If only one plot is called, the metric performs a comparison with a random labeling. If two plots are called, the labeling of the first is compared with the labeling of the second. If the "Monte Carlo" option is chosen for any validity metric, that metric is run once with the current coloring and then several times with random colorings (number of iterations dependent on the particular metric). The histogram of values is then displayed, representing the probability density function for that metric, with the actual calculated value displayed as a red line.

The Close Plots button closes all open plot windows. Other dialogs, such as the cluster or validity dialog, remain open. (Closing the Controller window itself will close all child windows, including both plots and dialogs.)

The Tile button rearranges current plot windows so that all are visible at once. In the process, the Controller or some of its dialogs may be hidden beneath plots. Also, some plots may overlap the title bars of other plots.

3. SCRIPTS FOR VALIDITY TESTING

For convenience in command-line use of the tool, the various validity metrics return simple arrays of values. For example, if one were to attempt to find a reasonable value of k for k-means, the following code could be used to generate a plot of Davies-Bouldin vs. k for k between 5 and 15.

```
function [] = runDBvskmeans(filename)

[names, xcoords, ycoords] = textread(filename, ...
                                     '%s%f%f');
points = [xcoords, ycoords];
labels = zeros(size(names));
startRange = 1;
endRange = 15;
results = zeros(1, endRange);
for numClusters = startRange:endRange
    labels = kmeans(points, numClusters);
    results(numClusters) = ...
            DaviesBouldin(labels, points);
end
plot(results, [startRange:endRange]);
```

4. KNOWN ISSUES

There are currently only 78 color/symbol combinations available (91 if the color black is included).

When new plots are opened, they do not automatically

```
acquire the coloring of existing plots.  This means
that colorings may be inconsistent or misleading.  It
is currently the user’s responsibility to keep track
of this.

There is no test for legal values in the EPS or
Clusters fields in the Cluster dialog box.  EPS should
be a positive number and Clusters should be a positive
integer, but if you enter something else, it won’t
notify you of the error.

No progress notification is given while clustering
algorithms or validity metrics are running.  Though
these processes can take a number of minutes to run,
the user must patiently wait for them to complete.
The only methods for aborting a clustering or validity
metric are hitting Ctrl-C in the MATLAB command window
or closing MATLAB entirely.


5. LIST OF INCLUDED FILES

README
DaviesBouldin.m
DaviesBouldinWrapper.m
alt_kmeans.m
clustercontrol.m
controller.m
decodecolor.m
density.m
dist.m
encodecolor.m
fastcolor.m
gettoolplots.m
mergecolor.m
metrictable.m
modifiedHubertGamma.m
modifiedHubertGammaWrapper.m
multsinglelink.m
plotwindow.fig
plotwindow.m
protogui.fig
protogui.m
randDaviesBouldin.m
randMHG.m
randRandTypes.m
randTypes.m
randTypesWrapper.m
savecontrol.fig
savecontrol.m
selectcontrol.m
slowcolor.m
validitycontrol.fig
zoom.m
```

Appendix B

Source Code

Included here is all source code for the tool. Since some lines of code were wider than the width of this page, these have been wrapped onto multiple lines. Code continued from a previous line is marked by the letters "`(cont.)`" at the beginning of the line.

B.1 DaviesBouldin.m

```
%Find the Davies-Bouldin value for a given labeling.
(cont.)  See README
%for a better description of this metric.

function output = DaviesBouldin(labels, points)

numClusters = max(labels);
numPoints = length(labels);

%Find the center of each cluster

clusterPoints = ones(numClusters, 1);
xcenters = zeros(numClusters, 1);
ycenters = zeros(numClusters, 1);

for i = 1:numPoints
    current = labels(i);
    clusterPoints(current) = clusterPoints(current) +
(cont.)  1;
```

```
    xcenters(current) = xcenters(current) + points(i,
(cont.)  1);
    ycenters(current) = ycenters(current) + points(i,
(cont.)  2);
end

xcenters = xcenters ./ clusterPoints;
ycenters = ycenters ./ clusterPoints;

actualClusters = numClusters;
errors = zeros(numClusters, 1);
for i = 1:numClusters
    c = find(labels == i);
    if length(c) == 0
        actualClusters = actualClusters - 1;
    else
        for j = 1:length(c)
            c(j) = sqrt((points(c(j), 1) -
(cont.)  xcenters(i))^2 + ...
                        (points(c(j), 2) -
(cont.)  ycenters(i))^2);
        end
        errors(i) = var(c);
    end
end

DB = 0;

for i = 1:numClusters
    maxDB = 0;
    for j = 1:numClusters
        Rij = 0;
        denom = sqrt((xcenters(i) - xcenters(j))^2 +
(cont.)  ...
                     (ycenters(i) - ycenters(j))^2);
(cont.)
        if denom ~= 0
            Rij = (errors(i) + errors(j)) / denom;
(cont.)
        end
        if Rij > maxDB
            maxDB = Rij;
        end
    end
    DB = DB + maxDB;
end

output = DB / actualClusters;
```

B.2 DaviesBouldinWrapper.m

```
%Wrapper function to make printable strings.

function [output, retfig] = DaviesBouldinWrapper(
(cont.) (labels, points)

DB = DaviesBouldin(labels, points);
output = sprintf('Davies-Bouldin index: %d', DB);

% No new figures
retfig = [];
```

B.3 altkmeans.m

```
(cont.)                      %%%%
%k-means clustering, 10-22-02, ejwu
%
%Performs standard k-means clustering and returns a
(cont.)  column vector of labels
%    between 1 and k for each data point.
%
%Required arguments:
%    names should be an n x 1 column vector with the
(cont.)  ID's assigned to each
%        data point.  The size of names is assumed to
(cont.)  be the number of data
%        points.
%    points should be an n x 2 matrix with the x and y
(cont.)  coordinates for each
%        data point.  It had better be the same size n
(cont.)  as names, or all hell
%        probably breaks loose.
%    k is the number of clusters wanted.  Make it
(cont.)  positive.
%
%Optional arguments:
%    iterations: At most, k-means will run for this
(cont.)  many iterations.  Defaults
%        to 100 if negative input or no input is
(cont.)  given.
%    kthreshold: If the total change in all cluster
(cont.)  centers falls below
%        kthreshold, clustering is considered complete
(cont.)  and stops.  Defaults to 0
%        if no input or bad input is given.

function labels = alt_kmeans(names, points, k,
(cont.)  iterations, kthreshold)

if(nargin < 3)
    disp(sprintf('Need at least names, points, and k
(cont.)  as arguments.’));
    return;
end
if(k <= 0)
    disp(sprintf(’k must be positive, no clustering
(cont.)  done.’));
    return;
end
if(((nargin >= 4) & (iterations < 0)) | (nargin <
(cont.)  4))
    disp(sprintf(’No input or bad input given,
(cont.)  iterations defaulting to 100’));
    iterations = 100;
end
if(((nargin >= 5) & (kthreshold < 0)) | (nargin <
(cont.)  5))
    disp(sprintf(’No input or bad input given,
(cont.)  kthreshold defaulting to 0’));
    kthreshold = 0;
end

%Create three n x 1 arrays from the information in the
(cont.)  file
%[names, xcoords, points(:, 2)] = textread(filename,
(cont.)  ’%s%f%f’);

%Coordinates of the centers of the clusters
centers = zeros(k, 2);

%Number of data points
numItems = length(names);

%Randomly assign cluster centers somewhere within the
(cont.)  grid
for i = 1:k
    centers(i, 1) = rand(1) * (max(points(:, 1)) -
(cont.)  min(points(:, 2))) ...
                               + min(points(:, 1));
    centers(i, 2) = rand(1) * (max(points(:, 2)) -
(cont.)  min(points(:, 2))) ...
```

```
                               + min(points(:, 2));
end

%Initialize labels to be returned
labels = zeros(numItems, 1);

for its = 1:iterations

    %assign each point to the nearest cluster
    for i = 1:numItems
    %    labels(j) = closest(points(j, 1), points(j,
(cont.)  2), centers);
    nearest = 99999;
        for j = 1:length(centers)
            distance = sqrt((centers(j, 1) - points(i,
(cont.)  1))ˆ2 + ...
                            (centers(j, 2) - points(i,
(cont.)  2))ˆ2);
            if distance < nearest
                labels(i) = j;
                nearest = distance;
            end
        end
    end

    %recompute cluster centers
    %totalx, totaly, numitems
    clusters = zeros(k, 3);
    for i = 1:numItems
        clusters(labels(i), 1) = clusters(labels(i),
(cont.)  1) + points(i, 1);
        clusters(labels(i), 2) = clusters(labels(i),
(cont.)  2) + points(i, 2);
        clusters(labels(i), 3) = clusters(labels(i),
(cont.)  3) + 1;
    end

    for i = 1:k
        if clusters(i, 3) == 0
            centers(i, 1) = rand(1) * (max(points(:,
(cont.)  1)) - min(points(:, 1))) + min(points(:, 1));
            centers(i, 2) = rand(1) * (max(points(:,
(cont.)  2)) - min(points(:, 2))) + min(points(:, 2));
            disp(sprintf('Empty cluster, randomly
(cont.)  reassigning cluster center'));
        else
            centers(i, 1) = clusters(i, 1) /
(cont.)  clusters(i, 3);
            centers(i, 2) = clusters(i, 2) /
(cont.)  clusters(i, 3);
        end
    end

end

centers

% We'll let the caller do the plotting.  -DLowd
% TODO: return centers and plot those, too?

%scatter(points(:, 1), points(:, 2));
%hold on;
%scatter(centers(1:k, 1), centers(1:k, 2), 'r*');
```

B.4 clustercontrol.m

```
function varargout = clustercontrol(varargin)
% CLUSTERCONTROL Application M-file for
(cont.)  clustercontrol.fig
%    FIG = CLUSTERCONTROL launch clustercontrol GUI.
%    CLUSTERCONTROL('callback_name', ...) invoke the
(cont.)  named callback.
%
% The Cluster control is a dialog box that enables
(cont.)  users to cluster
% any open plot using any of the available
(cont.)  algorithms.

% Last Modified by GUIDE v2.0 07-Feb-2003 14:08:14

if nargin == 0  % LAUNCH GUI

    % See INIT - this should never be reached

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION
(cont.)  OR CALLBACK

    try
        if (nargout)
            [varargout{1:nargout}] =
(cont.)  feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
    catch
        disp(lasterr);
    end

end


%------------------------------------------------------
function fig = init(parent, plotnames)
% INIT
% Create a new Cluster control dialog with the given
(cont.)  parent controller
% and set of plot names to select from.

fig = openfig(mfilename,'reuse');

% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor

% Generate a structure of handles to pass to
(cont.)  callbacks, and store it.
handles = guihandles(fig);
handles.parent = parent;
guidata(fig, handles);
updatePlotNames(fig, plotnames);


% ------------------------------------------------------
function varargout = clustergo_Callback(h, eventdata,
(cont.)  handles, varargin)
% CLUSTERGO_CALLBACK
% Called when 'Cluster' button is clicked.  Asks
(cont.)  parent Controller
% to perform a cluster on the specified plot using the
(cont.)  given parameters,
% read from the GUI controls.

handles = guidata(h);
plotindex = get(handles.popupmenu1, 'Value');
eps = str2num(get(handles.eps, 'String'));
numclusters = str2num(get(handles.clust, 'String'));
clusteralg = get(handles.algorithmpopup, 'Value');
controller('activecluster', handles.parent, plotindex,
(cont.)  ...
            eps, numclusters, clusteralg);
guidata(h, handles);


% ------------------------------------------------------
function varargout = figure1_DeleteFcn(h, eventdata,
(cont.)  handles, varargin)
% FIGURE1_DELETEFCN
% Automatically called when the dialog is closed.
(cont.)  Notifies the parent
% controller.

controller('clusterClosingCallback', handles.parent);

% ----------------------------------------------------
function updatePlotNames(h, newplotnames)
% UPDATEPLOTNAMES
% Update the list of available plots to cluster with a
(cont.)  new list of plot
% names.  If no plots are available, display '<none>'
(cont.)  and disable the
% "Cluster" button.

handles = guidata(h);

% Provide a default behavior if no plots are open.
if (isempty(newplotnames)),
    enabled = 'off';
    newplotnames = '<none>';
else
    enabled = 'on';
end

set(handles.popupmenu1, 'String', newplotnames);
set(handles.clustergo, 'Enable', enabled);
```

B.5 controller.m

```
function varargout = controller(varargin)
% CONTROLLER Application M-file for controller.fig
%    FIG = CONTROLLER launch controller GUI.
%    CONTROLLER('callback_name', ...) invoke the named
(cont.)  callback.
%
% The Controller is the base application window.  It
(cont.)  contains buttons for
% executing all tasks within the application, and when
(cont.)  closed, it closes
% all child windows.

% Last Modified by GUIDE v2.0 11-Mar-2003 00:37:17

if nargin == 0  % LAUNCH GUI

    fig = openfig(mfilename,'reuse');

    % Due to our use of a global variable, we may only
(cont.)  have one controller
    % open at once.
    if (isglobal('CLUSTERTOOLFIG')),
        disp('Closing current controller to open a new
(cont.)  one.');
        close(CLUSTERTOOLFIG);
    end

    % Initialize the global reference to the
(cont.)  controller.  This allows
    % us to access the controller from the command
(cont.)  line without knowing
    % its figure number.
    global CLUSTERTOOLFIG;
    CLUSTERTOOLFIG = fig;

    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundC
    % Generate a structure of handles to pass to
(cont.)  callbacks, and store it.
    handles = guihandles(fig);

    % Color of points in a newly loaded plot
    handles.baseColor = 0;

    % Color last selected in Select dialog, for
(cont.)  manually recoloring
    % portions of plots.
    handles.selectColor = 0;

    % Array of all currently open plots.
    handles.plotnames = {};
    handles.plothandles = [];

    % Array index of plot window most recently clicked
(cont.)  on by user
    handles.activePlotIndex = 0;

    % Other child GUIs
    handles.clusterfig = 0;
    handles.selectfig = 0;
    handles.validityfig = 0;

    % List of other figures to be closed with the
(cont.)  controller
    handles.otherfigs = [];

    % Cache of pairwise distances
    handles.cachedIndex = 0;
    handles.cachedDistances = 0;

    % Current path, for opening new files
    handles.pathname = pwd;

    % Update handles structure
    guidata(fig, handles);

    if nargout > 0
```

```
        varargout{1} = fig;
    end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION
(cont.)  OR CALLBACK

    try
        if (nargout)
            [varargout{1:nargout}] =
(cont.)  feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
    catch
        disp(lasterr);
    end
end

%| ABOUT CALLBACKS:
%| GUIDE automatically appends subfunction prototypes
(cont.)  to this file, and
%| sets objects’ callback properties to call them
(cont.)  through the FEVAL
%| switchyard above. This comment describes that
(cont.)  mechanism.
%|
%| Each callback subfunction declaration has the
(cont.)  following form:
%| <SUBFUNCTION_NAME>(H, EVENTDATA, HANDLES,
(cont.)  VARARGIN)
%|
%| The subfunction name is composed using the object’s
(cont.)  Tag and the
%| callback type separated by ’_’, e.g.
(cont.)  ’slider2_Callback’,
%| ’figure1_CloseRequestFcn’, ’axis1_ButtondownFcn’.
%|
%| H is the callback object’s handle (obtained using
(cont.)  GCBO).
%|
%| EVENTDATA is empty, but reserved for future use.
%|
%| HANDLES is a structure containing handles of
(cont.)  components in GUI using
%| tags as fieldnames, e.g. handles.figure1,
(cont.)  handles.slider2. This
%| structure is created at GUI startup using
(cont.)  GUIHANDLES and stored in
%| the figure’s application data using GUIDATA. A copy
(cont.)  of the structure
%| is passed to each callback.  You can store
(cont.)  additional information in
%| this structure at GUI startup, and you can change
(cont.)  the structure
%| during callbacks.  Call guidata(h, handles) after
(cont.)  changing your
%| copy to replace the stored original so that
(cont.)  subsequent callbacks see
%| the updates. Type "help guihandles" and "help
(cont.)  guidata" for more
%| information.
%|
%| VARARGIN contains any extra arguments you have
(cont.)  passed to the
%| callback. Specify the extra arguments by editing
(cont.)  the callback
%| property in the inspector. By default, GUIDE sets
(cont.)  the property to:
%| <MFILENAME>(’<SUBFUNCTION_NAME>’, gcbo, [],
(cont.)  guidata(gcbo))
%| Add any extra arguments after the last argument,
(cont.)  before the final
%| closing parenthesis.


% ------------------------------------------------------
% OPEN SECTION: code related to the Open button and
(cont.)  opening new plots.
% ------------------------------------------------------
% -----------------------------------------------------
function varargout = openbutton_Callback(h, eventdata,
(cont.)  handles, varargin)

% Save workplace directory, then switch to working
(cont.)  dir
currdir = pwd;
cd(handles.pathname);

% Get filename from the user
[filename, newpath] = ...
    uigetfile({'*.coord', 'Ordinations (*.coord)';
(cont.)  '*.*', 'All Files (*.*)'}, ...
        'Open File');

% Restore workplace directory
cd(currdir);

% If the user hit 'Cancel' then we should abort
if (isequal(filename,0)),
    return;
end

% Get points
[names, xcoords, ycoords] = textread([newpath,
(cont.) ,filename], '%s%f%f');
points = [xcoords, ycoords];

% Initial coloring: all the same color
[height, width] = size(points);
colors = zeros(height, 1) + handles.baseColor;

f = plotwindow('init', h, filename, names, points,
(cont.)  colors);

% Set the plot to (approximately) its own size and
(cont.)  shape.  Weird hack so that
% tile works right the first time.  I have no clue why
(cont.)  this is necessary.
set(f, 'Position', get(f, 'Position') + [1,1,1,1]);
```

```
% Update handles...

% First, make sure our copy is up-to-date
handles = guidata(h);

% Then, add this plot, its name, and set the path
handles.plothandles = [handles.plothandles, f];
handles.plotnames = {handles.plotnames{:}, filename};
handles.pathname = newpath;

% Finally, save changes
guidata(h, handles);


% Set initial button-down behavior for the plot
if (handles.selectfig == 0),
    enableZoom(f);
else
    enableSelect(f);
end;

% Enable saving/loading of colorings
set(handles.savebutton, ’Enable’, ’on’);
set(handles.loadbutton, ’Enable’, ’on’);


% Ensure that the zoom limits are set before we change
(cont.)  anything
% (i.e. by the Zoom to Cluster command)
zoom(f, ’getlimits’);

% Make this newly opened plot active
% (This will also cause the list of plot names to be
(cont.)  updated.)
setActivePlot(h, f);


% --------------------------------------------------
% CLUSTER SECTION: code related to the Cluster button
```

```
(cont.)  and callbacks
% related to its dialog.
% ----------------------------------------------------

% ----------------------------------------------------
function varargout = clusterbutton_Callback(h,
(cont.)  eventdata, handles, varargin)
% CLUSTERBUTTON_CALLBACK
% Callback function for the Cluster toggle button.
(cont.)  Creates/destroys
% the Cluster dialog box.

if (get(handles.clusterbutton, ’Value’) == 1),
    % Open cluster GUI
    handles.clusterfig = clustercontrol(’init’, h,
(cont.)  []);
    updatePlotNames(h, handles);
else,
    close(handles.clusterfig);
    handles.clusterfig = 0;
end

guidata(h, handles);


% ----------------------------------------------------
function varargout = activecluster(h, plotindex, eps,
(cont.)  numclusters, clusteralg)
% ACTIVECLUSTER
% Cluster data in a specific plot according to the
(cont.)  algorithm and parameters
% specified.  Recolors all plots to reflect the
(cont.)  clustering.  Generally called
% by the cluster dialog box.

handles = guidata(h);

% The first plot name is that of the active plot.
(cont.)  Therefore, all subsequent
% names are off by one.
```

```
plotindex = plotindex - 1;
if (plotindex == 0),
    plotindex = handles.activePlotIndex;
end

% DEBUG
disp([’Clustering plot: ’,handles.plotnames{plotindex}

[filename, names, points, colors] = ...
    plotwindow(’getInfo’, handles.plothandles(
(cont.) (plotindex));

% This is a temporary kludge since iterations aren’t
(cont.)  being used and eps is.
iterations = 100;

% K-means clustering
if (clusteralg == 1),

    % If we’re using version 13 or later, use the
(cont.)  version of kmeans from
    % the stats package.  Otherwise, run Eric’s
(cont.)  implementation, since
    % the R12 stats package doesn’t include a kmeans
(cont.)  implementation.
    if (sscanf(version(’-release’), ’%f’) > 12),
        clusters = kmeans(points, numclusters,
(cont.)  ’Start’, ’cluster’, ’Replicates’, 3, ...
            ’Maxiter’, iterations, ’EmptyAction’,
(cont.)  ’singleton’);
    else,
        clusters = alt_kmeans(names, points,
(cont.)  numclusters, iterations);
    end

% All the linkage clustering algorithms
elseif (clusteralg < 7),
    if (plotindex == handles.cachedIndex),
        distances = handles.cachedDistances;
    else
```

```
            distances = pdist(points);

            % Save newly computed results
            handles.cachedDistances = distances;
            handles.cachedIndex = plotindex;
            guidata(h, handles);
        end

        linkageTypes = {’single’, ’average’, ’complete’,
(cont.)  ’centroid’, ’ward’};
        linky = linkage(distances,
(cont.)  linkageTypes{clusteralg-1});
        clusters = cluster(linky, numclusters);

elseif (clusteralg == 7),

        clusters = density(names, points, eps);

        % ...Insert additional algorithms here...

else
        disp([’Unknown algorithm: ’,
(cont.)  int2str(clusteralg)]);
end

% Change the cluster numbers so we have no black
(cont.)  clusters.
% It’s easier to see what’s been clustered and what
(cont.)  hasn’t if all
% black points are not in clusters.  Then, subtract
(cont.)  one, because our
% numbering scheme starts with 0, not 1.
adjClusters = clusters + ceil(clusters/6) - 1;

% Color noise clusters (cluster 0) as black points.
adjClusters(find(clusters == 0)) = 0;

% Recolor all plots appropriately.
for j = handles.plothandles,
    plotwindow(’recolor’, j, names, adjClusters);
```

```
end;

% DEBUG
disp('Clustering complete.');

% ----------------------------------------------------
function clusterClosingCallback(h)
% CLUSTERCLOSINGCALLBACK
% Untoggle cluster button when its dialog is closed.

handles = guidata(h);
set(handles.clusterbutton, 'Value', 0);
handles.clusterfig = 0;
guidata(h, handles);

% ----------------------------------------------------
% SELECT SECTION: code related to the Select button
(cont.)  and callbacks
% related to its dialog.
% ----------------------------------------------------

% ----------------------------------------------------
function varargout = selectbutton_Callback(h,
(cont.)  eventdata, handles, varargin)
% SELECTBUTTON_CALLBACK
% Opens the select GUI, used to pick colors for
(cont.)  manually coloring plots,
% or if already open, closes it.

if (get(handles.selectbutton, 'Value') == 1)
    % Open selection GUI
    handles.selectfig = selectcontrol('init', h,
(cont.)  handles.selectColor);

    % Switch the mode of each plot window
    disableZoom(handles.plothandles);
    enableSelect(handles.plothandles);
else
    % Switch back to the default zoom mode
    disableSelect(handles.plothandles);
    enableZoom(handles.plothandles);

    close(handles.selectfig);
    handles.selectfig = 0;
end

guidata(h, handles);

% -----------------------------------------------------
function setSelectColor(h, colorIndex)
% SETSELECTCOLOR
% Set the color to be used when manually recoloring
(cont.)  points on a plot,
% in select mode.  Called by the selectcontrol GUI.

handles = guidata(h);
handles.selectColor = colorIndex;
guidata(h, handles);

% -----------------------------------------------------
function colorIndex = getSelectColor(h)
% GETSELECTCOLOR
% Get the color to be used when manually recoloring
(cont.)  points on a plot,
% in select mode.  Called by the plotwindow GUI.

handles = guidata(h);
colorIndex = handles.selectColor;

% -----------------------------------------------------
function plotPointSelectionCallback(h, names)
% PLOTPOINTSELECTIONCALLBACK
% Recolor the points with the given names to the
(cont.)  current select color
% in all plots.  Called by the plotwindow GUI when in
(cont.)  select mode.

handles = guidata(h);

for fig = handles.plothandles,
    plotwindow('select', fig, names,
(cont.)  handles.selectColor);
end;

figure(handles.plothandles(handles.activePlotIndex));

% ----------------------------------------------------
function selectClosingCallback(h)
% SELECTCLOSINGCALLBACK
% Switch out of selection mode when required GUI is
(cont.)  closed.

handles = guidata(h);

% Untoggle Select button
set(handles.selectbutton, 'Value', 0);

% Select GUI no longer exists
handles.selectfig = 0;

% Reenable zoom
disableSelect(handles.plothandles);
enableZoom(handles.plothandles);

guidata(h, handles);


% ----------------------------------------------------
function varargout = zoomToCluster(h, color)
% ZOOMTOCLUSTER
% Zoom all plots appropriately, to the cluster of the
(cont.)  specified color index
% Called when user Shift-Clicks on a plotwindow.

handles = guidata(h);


currfig = gcf;

% Forward request to each plotwindow; they know how to
```

```
(cont.)  zoom themselves.
for f = handles.plothandles,
    plotwindow('zoomToCluster', f, color);
end

figure(currfig);

% -----------------------------------------------------
function setActivePlot(h, fig)
% SETACTIVEPLOT
% Record that the specified figure is the currently
(cont.)  active plotwindow.

handles = guidata(h);
handles.activePlotIndex = find(handles.plothandles ==
(cont.)  fig);

guidata(h, handles);
updatePlotNames(h, handles);



% -----------------------------------------------------
% CLEAR SECTION: code related to the Clear Colors
(cont.)  button.
% -----------------------------------------------------

function varargout = clearbutton_Callback(h,
(cont.)  eventdata, handles, varargin)
% CLEARBUTTON_CALLBACK
% Clear the colors from all plots, except those with
(cont.)  'Hold Colors' on.

for f = handles.plothandles,
    plotwindow('clearColors', f);
end


% -----------------------------------------------------
% SAVE SECTION: code related to the Save Colors button
```

```
(cont.)  and saving a
% plot’s coloring.
% ----------------------------------------------------

% ----------------------------------------------------
function varargout = savebutton_Callback(h, eventdata,
(cont.)  handles, varargin)
% SAVEBUTTON_CALLBACK
% Opens the save colors GUI

savecontrol(’init’, h, handles.pathname,
(cont.)  char(handles.plotnames));

% ----------------------------------------------------
function varargout = saveColorsCallback(h, plotIndex,
(cont.)  pathname, filename)
% SAVECOLORSCALLBACK
% Write the colors of the specified plot to the given
(cont.)  filename.  This
% method should be called from the save colors dialog
(cont.)  box, one a plot
% and filename have been selected.

handles = guidata(h);

% Get plot info
[fname, names, points, colors] = ...
    plotwindow(’getInfo’, handles.plothandles(
(cont.) (plotIndex));

% Write file
fid = fopen([pathname,filename], ’w+’);
for i = 1:length(colors),
    fprintf(fid, ’%s\t%d\n’, names{i}, colors(i));
end
fclose(fid);

% Save the new default file path
handles.pathname = pathname;
guidata(h, handles);
```

```
% -----------------------------------------------------
% LOAD SECTION: code related to the Load button and
(cont.)  loading colorings.
% -----------------------------------------------------

% -----------------------------------------------------
function varargout = loadbutton_Callback(h, eventdata,
(cont.)  handles, varargin)
% LOADBUTTON_CALLBACK
% Asks the user to interactively select a coloring
(cont.)  from a file, which it
% loads and applies to all open plots whose colors are
(cont.)  not being held.

% Save workplace directory, then switch to working
(cont.)  dir
currdir = pwd;
cd(handles.pathname);

% Get filename from the user
[filename, handles.pathname] = ...
    uigetfile({'*.color', 'Coloring (*.color)'; '*.*',
(cont.)  'All Files (*.*)'}, ...
        'Load Coloring');

% Restore workplace directory
cd(currdir);

if (isequal(filename,0))
    return;
end

% Read file
[names, colors] = textread([handles.pathname,filename],
(cont.) , '%s%f');

% Recolor all plots appropriately
```

```
for f = handles.plothandles,
    plotwindow('recolor', handles.plothandles(f),
(cont.)  names, colors);
end;


% ----------------------------------------------------
% VALIDITY SECTION: code related to the Validity
(cont.)  button and running
% comparison validity metrics on different plots.
% ----------------------------------------------------

% ----------------------------------------------------
function varargout = validitybutton_Callback(h,
(cont.)  eventdata, handles, varargin)
% VALIDITYBUTTON_CALLBACK
% Callback function for the Validity toggle button.
% Creates/destroys the validity dialog box.

if (get(handles.validitybutton, 'Value') == 1)
    % Open validity GUI
    handles.validityfig = validitycontrol('init', h);
    updatePlotNames(h, handles);
else
    close(handles.validityfig);
    handles.validityfig = 0;
end

guidata(h, handles);


% ----------------------------------------------------
function result = validityCallback(h, method,
(cont.)  plotindex1, plotindex2)
% VALIDITYCALLBACK
% Runs the specified validity metric for the plots of
(cont.)  the given indices, and
% returns a string with whatever output the metric
(cont.)  produced.  If two plots
% are being compared to each other, their data will
```

```
(cont.)  first be restricted to
% the points they have in common.

handles = guidata(h);

% Default to one-plot case when second plot index
(cont.)  isn’t passed.
if (nargin == 3),
    plotindex2 = 0;
end

[filename1, names1, points1, colors1] = ...
    plotwindow(’getInfo’, handles.plothandles(
(cont.) (plotindex1));

% Single-plot case (most common)
if (plotindex2 == 0),
    [result, newfigs] = feval(method, colors1,
(cont.)  points1);

% Two-plot case (more complicated...)
else,
    [filename2, names2, points2, colors2] = ...
        plotwindow(’getInfo’,
(cont.)  handles.plothandles(plotindex2));

    % Compare plots based only on their common
(cont.)  points.
    [commonNames, indices1, indices2] =
(cont.)  intersect(names1, names2);

    % For this to work, the lists must be ordered the
(cont.)  same as well.
    [sortNames, sort1] = sort(names1(indices1));
    [sortNames, sort2] = sort(names2(indices2));

    % Therefore, we reorder the common points indices
(cont.)  so they’ll both
    % represent the same order.
    i1 = indices1(sort1);
    i2 = indices2(sort2);

    % Finally, call the actual metric.
    [result, newfigs] = feval(method, colors1(i1),
(cont.)  points1(i1), ...
                               colors2(i2),
(cont.)  points2(i2));
end

for f = newfigs,
    set(f, 'DeleteFcn', 'controller(
(cont.) (''figClosingCallback'', gcbo)');
end
handles.otherfigs = [handles.otherfigs, newfigs];
guidata(h, handles);


% ------------------------------------------------------
function validityClosingCallback(h)
% VALIDITYCLOSINGCALLBACK
% Untoggle validity button when its dialog is closed.

handles = guidata(h);

% Untoggle validity button
set(handles.validitybutton, 'Value', 0);

% Signal that the dialog is no longer open, so we
(cont.)  won't try closing
% it later or sending it updates.
handles.validityfig = 0;
guidata(h, handles);


% ------------------------------------------------------
% CLOSE SECTION: callback for the Close Plots button.
% ------------------------------------------------------

% ------------------------------------------------------
function varargout = closebutton_Callback(h,
(cont.)  eventdata, handles, varargin)
% CLOSEBUTTON_CALLBACK
% Callback for the Close Plots button.  Closes all
(cont.)  open plots.

% Iterate through all open plots and close them one by
(cont.)  one.
% Resetting the active plot index and notification of
(cont.)  appropriate dialogs
% is all taken care of by the window closing callbacks
(cont.)  made by the plots
% themselves; therefore, we needn’t worry about that
(cont.)  here.  Just closing
% them will do the right thing.
for f = handles.plothandles,
    close(f);
end


% ------------------------------------------------------
% TILE SECTION: callback for the Tile button.
% ------------------------------------------------------

% ------------------------------------------------------
function varargout = tilebutton_Callback(h, eventdata,
(cont.)  handles, varargin)
% TILEBUTTON_CALLBACK
% Callback function for the tile button.
% Arranges open plot windows so all are visible on the
(cont.)  screen at once.

handles = guidata(h);

numPlots = length(handles.plothandles);
if (numPlots < 1),
    return;
end

n = ceil(sqrt(numPlots));
```

```
screenDims = get(0, ’ScreenSize’);
totalWidth = screenDims(3);
totalHeight = screenDims(4);

w = totalWidth/n;
h = totalHeight/n;

for i = 1:numPlots,
    xIndex = mod(i-1, n);
    yIndex = (i-1 - xIndex)/n;
    x = floor(w * xIndex) + 1;
    y = floor(h * yIndex) + 1;
    set(handles.plothandles(i), ’Position’,
(cont.)  [x,y,w,h]);
end

% ---------------------------------------------------
function varargout = figure1_DeleteFcn(h, eventdata,
(cont.)  handles, varargin)
% FIGURE1_DELETEFCN
% Window deletion callback for the Controller window;
(cont.)  automatically
% called as the program exits.  Ensures that all child
(cont.)  windows are
% closed and appropriate globals are cleared.

% Close all open windows.
for f = handles.plothandles,
    close(f);
end

for f = handles.otherfigs,
    close(f);
end

if (handles.clusterfig ˜= 0),
    close(handles.clusterfig);
end

if (handles.validityfig ˜= 0),
    close(handles.validityfig);
end

if (handles.selectfig ˜= 0),
    close(handles.selectfig);
end

clear global CLUSTERTOOLFIG;

% -----------------------------------------------------
% UTIL SECTION: miscellaneous internal utility
(cont.)  functions
% -----------------------------------------------------

% -----------------------------------------------------
function enableZoom(plothandles)
% ENABLEZOOM
% Enable zoom mode for specified plots.  (To switch
(cont.)  from select to zoom
% mode for a plot, first call DISABLESELECT, then
(cont.)  ENABLEZOOM.)

for f = plothandles,
    plotwindow('enableZoom', f);
end

% -----------------------------------------------------
function disableZoom(plothandles)
% DISABLEZOOM
% Disable zoom mode for specified plots.  (To switch
(cont.)  from zoom to
% select mode for a plot, first call DISABLEZOOM, then
(cont.)  ENABLESELECT.)

for f = plothandles,
    plotwindow('disableZoom', f);
end

% -----------------------------------------------------
function enableSelect(plothandles)
```

```
% ENABLESELECT
% Enable select mode for the specified plots, to
(cont.)  permit manual clustering.
% (To switch from zoom to select mode for a plot,
(cont.)  first call DISABLEZOOM,
% then ENABLESELECT.)

for f = plothandles,
    plotwindow('enableSelect', f);
end

% -----------------------------------------------------
function disableSelect(plothandles)
% DISABLESELECT
% Disable select mode for the specified plots.  (To
(cont.)  switch from select to
% zoom mode for a plot, first call DISABLESELECT, then
(cont.)  ENABLEZOOM.)

for f = plothandles,
    plotwindow(f, 'disableSelect');
end

% -----------------------------------------------------
function updatePlotNames(h, handles)
% UPDATEPLOTNAMES
% Notify cluster and/or validity dialogs of the new
(cont.)  set of plot names.
% This is important because which plots are open may
(cont.)  change, and both
% dialogs let the user select from currently available
(cont.)  plots for
% clustering or comparison.

% Notify cluster dialog
if (handles.clusterfig ~= 0),
    if (length(handles.plothandles) == 0),
        clustercontrol('updatePlotNames',
(cont.)  handles.clusterfig, []);
    else,
        % The cluster dialog also needs to know the
(cont.)  name of the active plot
        activePlotName = ['Active Plot (', ...
                char(handles.plotnames(
(cont.) (handles.activePlotIndex)), ')'];
        names = strvcat(activePlotName,
(cont.)  char(handles.plotnames));
        clustercontrol('updatePlotNames',
(cont.)  handles.clusterfig, names);
    end
end

% Notify validity dialog
if (handles.validityfig ~= 0),
    validitycontrol('updatePlotNames',
(cont.)  handles.validityfig, ...
        char(handles.plotnames));
end

% ------------------------------------------------------
function varargout = plotClosingCallback(h, plotfig)
% PLOTCLOSINGCALLBACK
% Callback made by plot windows whenever closed.
(cont.)  Allows the controller to
% remove the calling plot from its list of managed
(cont.)  plots, and maintain a
% consistent state in general.

handles = guidata(h); % get handle data back again

plotIndex = find(handles.plothandles == plotfig);

if (isempty(plotIndex)),
    disp(['ERROR: plot figure ', num2str(plotfig), '
(cont.)  not found!']);
    disp('Current state:');
    disp(handles.plothandles);
end

% Update the activePlotIndex
```

```
if (handles.activePlotIndex == plotIndex),
    handles.activePlotIndex = 1;
elseif (handles.activePlotIndex > plotIndex)
    handles.activePlotIndex = handles.activePlotIndex
(cont.)  - 1;
end

% Clear cache or update index of cached plot
if (handles.cachedIndex == plotIndex),
    handles.cachedIndex = 0;
    handles.cachedDistances = 0;
elseif (handles.cachedIndex > plotIndex),
    handles.cachedIndex = handles.cachedIndex - 1;
end

% Keep all plots except the one with the specified
(cont.)  figure handle
remainingPlots = find(handles.plothandles ˜=
(cont.)  plotfig);
handles.plotnames = {handles.plotnames{remainingPlots}
handles.plothandles = handles.plothandles(
(cont.) (remainingPlots);
updatePlotNames(h, handles);

% If no plots are currently open, disable the
(cont.)  save/load colors buttons
if (length(handles.plothandles) == 0),
    set(handles.savebutton, 'Enable', 'off');
    set(handles.loadbutton, 'Enable', 'off');
end

guidata(h, handles); % save changes to handles

% ----------------------------------------------------
function varargout = figClosingCallback(fighandle)
% FIGCLOSINGCALLBACK
% Callback made by other, extra figures whenever
(cont.)  closed.  Allows the
% controller to remove the calling figure from its
(cont.)  list of managed figures.
```

```
% These managed figures are generally created by
(cont.)  validity metrics, but
% should be closed when the controller is.

% HACK: use of global variable
global CLUSTERTOOLFIG;
h = CLUSTERTOOLFIG;
handles = guidata(h); % get handle data back again

remainingfigs = find(handles.otherfigs ~= fighandle);
handles.otherfigs = handles.otherfigs(remainingfigs);
guidata(h, handles);

% ------------------------------------------------------
function plothandles = getplothandles(h)
% GETPLOTHANDLES
% Returns a list of handles to all plot windows
(cont.)  managed by this
% controller.

handles = guidata(h);
plothandles = handles.plothandles;
```

B.6 decodecolor.m

```
function [color, symbol] = decodecolor(colornum)
% DECODECOLOR
% Determine the color and symbol from the numerical
(cont.)  index specified.
% The returned color and symbol are integers from 1-7
(cont.)  and 1-13 respectively.

numColors = 7;
numSymbols = 13;

% Convert color number to a specific color and symbol
(cont.)  for plotting.
color = mod(colornum, numColors) + 1;
symbol = mod(floor(colornum/numColors), numSymbols) +
(cont.)  1;
```

B.7 density.m

```
%%%%
%density clustering, 4.1.03 avani
%
%Performs standard density-based clustering as
(cont.)  described in "A Density-Based
%Algorithm for Discovering Clusters in Large Spatial
(cont.)  Databases with Noise" by
%Martin Ester, et al.
%
% Required arguments:
%    names should be an n x 1 column vector with the
(cont.)  ID's assigned to each
%        data point.  The size of names is assumed to
(cont.)  be the number of data
%        points.
%    points should be an n x 2 matrix with the x and y
(cont.)  coordinates for each
%        data point.  It had better be the same size n
(cont.)  as names, or all hell
%        probably breaks loose.
%
%    eps is basically how tight you want the density
(cont.)  to be.  The algorithm is
%    experimentally linear, so you could just test
(cont.)  different eps values for your
%    data set.  This is required because this value
(cont.)  completely depends on
%
% Optional arguments:
%  minPoints is the minimum number of points the must
(cont.)  be within one
%  eps-neighborhood of a point for them to be in the
(cont.)  same cluster-core.  This
%  is by default ceil(log_2 n)
%
%

function labels = density(names, points, eps,
(cont.)  minPoints)

if(nargin < 3)
    disp(sprintf('Need at least names, points, and eps
(cont.)  as arguments.'));
    return;
end

if(eps <= 0)
    disp(sprintf('eps must be positive, no clustering
(cont.)  done.'));
    return;
end

%Number of data points
numItems = length(names);

if(((nargin >= 4) & (minPoints < 0)) | (nargin < 4))
    %minPoints = ceil(log(numItems))
    minPoints = 5;
end

points = transpose(points);

%Initialize labels to be returned
labels = zeros(numItems, 1);

id = 1; % ID of initial cluster

% The structure is doubly linked, which hopefully
(cont.)  makes it O(n)

for its = 1:numItems
    if labels(its) == 0 % 0 means it hasn't been
(cont.)  classified yet

        lenseeds = 0;
        seeds=zeros(numItems,1);

      % This is to see what is in the eps neighborhood
```

```
(cont.)  of a point.  Note that
      % some of the labels are there so its easier to
(cont.)  convert to R* trees later.

        for i = 1:numItems
            if (abs(dist(points(:,i),points(:,its)))<=
(cont.)  eps)
            seeds(i)=1;
            lenseeds = lenseeds + 1;
        end
        end

      % The above for loop is decidedly non-optimal.
      % Use R* Trees for better performance, and have
(cont.)  fun coding them in MATLAB.

      if (lenseeds < minPoints)
          labels(its)= -1; % its Noise
      else
          for j=1:numItems
          if (seeds(j)>0)

      if (labels(j) == 0)
                  labels(j)=id; % Sets your Points
(cont.)  ID
              % Now, checking some neighbors...
(cont.)  This construction
  % will be much faster if you use R* trees.
              num = 0;

  % Checking density.
  for loop = 1:numItems
                          if
(cont.)  (abs(dist(points(:,loop),points(:,j)))<= eps)
                          result(loop)=1;
                          num = num + 1;
                      end
                      end

          if (num >= minPoints)
```

```
              for more= 1:minPoints
          % The labels < 1 covers UNCLASSIFIED and
(cont.)  NOISE
             if (result(more)==1)
                 if (labels(more) == 0)
                 seeds(more) = 1;
                 labels(more) = id;
                 end
             if (labels(more) == -1)
                 labels(more) = id;
                 end
             end
              end
          end
          end
      end
      end
      id = id + 1;
  end
end

for lastloop = 1:numItems
    % All points which I are NOISE are in cluster 0
    if (labels(lastloop)== -1)
     labels(lastloop)=0;
    end
end

end
```

B.8 dist.m

```
function x = dist(p1,p2)
x = sqrt((p1(1)-p2(1))^2+(p1(2)-p2(2))^2);
% Basic Distance Function
```

B.9 encodecolor.m

```
function colornum = encodecolor(color, symbol)
% ENCODECOLOR
% Convert the specified color and symbol pair to a
(cont.)  single integer
% representation.  Legal color values are 1-7; legal
(cont.)  symbol values are 1-13.

colornum = (symbol - 1) * 7 + color - 1;
```

B.10 fastcolor.m

```
function colors2 = fastcolor(names2, names1, colors1,
(cont.)  baseColors)
% Given a mapping of names to colors, assign colors to
(cont.)  another set of names.
% It is assumed that both sets of names are ordered
(cont.)  identically and that they
% are differ by contiguous subsets of at most k names.
(cont.)   These two assumptions
% allows us to do the coloring in O(nkˆ2).  However,
(cont.)  when these assumptions
% are violated, the algorithm will fail.

% k -- maximum number of names to search through
(cont.)  before assuming a given
% name is not present in the other.
MAX_OFFSET = 500;

colors2 = baseColors;

idx1 = 1;
idx2 = 1;

while (idx1 <= length(colors1) & idx2 <=
(cont.)  length(colors2)),
    if (idx1 == length(colors1)),
        a = 1;
    end
    for k = 0:MAX_OFFSET,
        % Ensure that our experimental offsets don’t
(cont.)  overflow the arrays
        if (idx1 + k > length(colors1)),
            k1 = length(colors1) - idx1;
        else,
            k1 = k;
        end

        if (idx2 + k > length(colors2)),
            k2 = length(colors2) - idx1;
        else,
            k2 = k;
        end

        for off1 = 0:k1,
            if strcmp(names1(idx1 + off1), names2(idx2
(cont.)  + k2))
                break;
            end
        end

        if strcmp(names1(idx1 + off1), names2(idx2 +
(cont.)  k2)),
            idx1 = idx1 + off1;
            idx2 = idx2 + k2;
            break;
        end

        for off2 = 0:k2,
            if strcmp(names1(idx1 + k1), names2(idx2 +
(cont.)  off2))
                break;
            end
        end

        if strcmp(names1(idx1 + k1), names2(idx2 +
(cont.)  off2)),
            idx1 = idx1 + k1;
            idx2 = idx2 + off2;
            break;
        end
    end

    if (k == MAX_OFFSET),
        disp('Reverting to slowcolor.');
        colors2 = slowcolor(names2, names1, colors1,
(cont.)  baseColors);
        break;
    end
```

```
            colors2(idx2) = colors1(idx1);
            idx1 = idx1 + 1;
            idx2 = idx2 + 1;
        end
```

B.11 genGaussianBlob.m

```
function [points] = genGaussianBlob(numPoints,
(cont.)  xcenter, ycenter, xvariance, yvariance, theta)

points = zeros(numPoints, 2);
theta = theta / (2 * pi);
points(:, 1) = sqrt(xvariance) * randn(1,
(cont.)  numPoints)’;
points(:, 2) = sqrt(yvariance) * randn(1,
(cont.)  numPoints)’;
points = points * [cos(theta) -sin(theta);sin(theta)
(cont.)  cos(theta)];
points(:, 1) = points(:, 1) + xcenter;
points(:, 2) = points(:, 2) + ycenter;

scatter(points(:, 1), points(:, 2), 10);
axis equal
```

B.12 gettoolplots.m

```
function plothandles = gettoolplots
% GETTOOLPLOTS
% Returns an array of handles to each plotwindow
(cont.)  currently open.

% This global holds the figure handle for the
(cont.)  Controller GUI.
global CLUSTERTOOLFIG;

if (isempty(CLUSTERTOOLFIG)),
    disp('No controller currently open.');
    plothandles = [];
else
    plothandles = controller('getplothandles',
(cont.)  CLUSTERTOOLFIG);
end
```

B.13 hubertGamma.m

```
function output = HubertGamma(labels, points)

p = squareform(pdist(points));
q = ones(length(points), length(points));
for i = 1:length(points)
    for j = 1:length(points)
        if labels(i) == labels(j)
            q(i, j) = 0;
        end
    end
end

sum = 0;

for i = 1:length(points) - 1
    for j = i + 1:length(points)
        sum = sum + p(i, j) * q(i, j);
    end
end

output = sprintf('Hubert Gamma = %d\n', sum)

iterations = 20;
randomValues = zeros(iterations, 1);

for z = 1:iterations

maxLabel = length(unique(labels));
labels = ceil(rand(size(labels)) * maxLabel);

for i = 1:length(points)
    for j = 1:length(points)
        if labels(i) == labels(j)
            q(i, j) = 0;
        end
    end
end
```

```
for i = 1:length(points) - 1
    for j = i + 1:length(points)
        randomValues(z) = randomValues(z) + p(i, j) *
(cont.)  q(i, j);
    end
end

end

f = figure(87);
hist(randomValues);
h = hist(randomValues);
%bar(h);
hold on
stem(sum, max(h), ’r:’);
%sprintf(’Hubert (random) = %d\n’, sum)
```

B.14 metrictable.m

```
function table = metrictable
% METRICTABLE
% Returns a table of all validity metrics and options.
(cont.)   The table
% is an Nx1 cell array of structs.  Each struct
(cont.)  represents a single
% metric with two fields: name and options.  name is a
(cont.)  string describing
% the overall metric.  options is an Mx3 cell array,
(cont.)  with each row
% representing a different possible sub-metric.  The
(cont.)  first options column
% is a string representing the name of the option, the
(cont.)  second is the
% number of arguments it takes (one or two plots), and
(cont.)  the third is the
% method to call.  At least one option must be
(cont.)  specified for any metric.

numMetrics = 3;
table = cell(numMetrics, 1);

table{1}.name = ’Rand/Jaccard/Fowlkes’;
table{1}.options = {’<none>’, 2, ’randTypesWrapper’;
                    ’Random label hypothesis’, 2,
(cont.)  ’randRandTypes’};

table{2}.name = ’Hubert Gamma’;
table{2}.options = {’<none>’, 1,
(cont.)  ’modifiedHubertGammaWrapper’;
                    ’Random label hypothesis’, 1,
(cont.)  ’randMHG’};

table{3}.name = ’Davies-Bouldin’;
table{3}.options = {’<none>’, 1,
(cont.)  ’DaviesBouldinWrapper’
                    ’Random label hypothesis’, 1,
(cont.)  ’randDaviesBouldin’};
```

```
% Sample table entry:
%
% table{2}.name = ’Dummy metric’;
% table{2}.options = {’First Option’, 2,
(cont.)  ’dummyMethod’;
%                    ’Second Option’, 1,
(cont.)  ’dummyMethod’};
```

B.15 modifiedHubertGamma.m

```
%Calculate the modified Hubert Gamma and normalized
(cont.)  modified
%Hubert Gamma statistics.

function output = modifiedHubertGamma(labels, points)

numClusters = max(labels);
numPoints = length(labels);
clusterPoints = ones(numClusters, 1);
xcenters = zeros(numClusters, 1);
ycenters = zeros(numClusters, 1);

%Find the center of each cluster

for i = 1:length(labels)
    current = labels(i);
    clusterPoints(current) = clusterPoints(current) +
(cont.)  1;
    xcenters(current) = xcenters(current) + points(i,
(cont.)  1);
    ycenters(current) = ycenters(current) + points(i,
(cont.)  2);
end

xcenters = xcenters ./ clusterPoints;
ycenters = ycenters ./ clusterPoints;

%P is the proximity matrix

P = squareform(pdist(points));      %distances
P = max(max(P)) - P;                %similarities

%QQ(i, j) is the distance between cluster i and j

QQ = zeros(numClusters);
for i = 1:numClusters
    for j = 1:numClusters
        QQ(i, j) = sqrt((xcenters(i) - xcenters(j))^2
(cont.)  + ...
                       (ycenters(i) -
(cont.)  ycenters(j))ˆ2);
    end
end

%Q(i, j) is the distance between the center of the
(cont.)  cluster to which
%point i belongs and the center of the cluster to
(cont.)  which point j
%belongs.

Q = zeros(size(P));

for i = 1:numPoints - 1
    for j = i + 1:numPoints
        Q(i, j) = QQ(labels(i), labels(j));
    end
end

hubert = 0;
normHubert = 0;

%Find the mean of the upper right diagonal of P and Q

sumP = 0;
sumQ = 0;
for i = 1:numPoints - 1
    for j = i + 1:numPoints
        sumP = sumP + P(i, j);
        sumQ = sumQ + Q(i, j);
    end
end

meanP = sumP / (numPoints * (numPoints - 1) / 2);
meanQ = sumQ / (numPoints * (numPoints - 1) / 2);
meanPsq = meanPˆ2;
meanQsq = meanQˆ2;

%Find the variance of the upper right diagonal of P
```

```
(cont.)  and Q

stdevP = 0;
stdevQ = 0;
for i = 1:numPoints - 1
    for j = i + 1:numPoints
        stdevP = stdevP + P(i, j)ˆ2 - meanPsq;
        stdevQ = stdevQ + Q(i, j)ˆ2 - meanQsq;
    end
end

stdevP = stdevP / (numPoints * (numPoints - 1) / 2);
stdevQ = stdevQ / (numPoints * (numPoints - 1) / 2);
varP = sqrt(stdevP);
varQ = sqrt(stdevQ);

for i = 1:numPoints - 1
    for j = i + 1:numPoints
        hubert = hubert + P(i, j) * Q(i, j);
        normHubert = normHubert + (P(i, j) - meanP) *
(cont.)  ...
                                  (Q(i, j) - meanQ);
    end
end



hubert = hubert / (numPoints * (numPoints - 1) / 2);
normHubert = normHubert / (numPoints * (numPoints - 1)
(cont.)  / 2);

normHubert = normHubert / (varP * varQ);

output = [hubert, normHubert];
```

B.16 modifiedHubertGammaWrapper.m

```
%Takes the MHG of the points and puts it in a nice
(cont.)  string to be
%printed in the Validity controller window.

function [output, retfig] = modifiedHubertGammaWrapper(
(cont.) (labels, points)

huberts = modifiedHubertGamma(labels, points);

output = sprintf('Modified Hubert Gamma = %f',
(cont.)  huberts(1));
output = strvcat(output, sprintf('Normalized Modified
(cont.)  Hubert Gamma = %f\n', ...
                                  huberts(2)));

% No new figures
retfig = [];
```

B.17 multsinglelink.m

```
function output = multsinglelink(filename, startrange,
(cont.)  endrange)

results = [];
[names, xcoords, ycoords] = textread(filename,
(cont.)  '%s%f%f');
points = [xcoords, ycoords];
distances = pdist(points);
linky = linkage(distances, 'single');

for numclusters = startrange : endrange,
    labels1 = cluster(linky, numclusters);

    hub1 = modifiedHubertGamma(labels1, points);

    results = [results; hub1];

end
 output = results;
```

B.18 plotwindow.m

```
function varargout = plotwindow(varargin)
% PLOTWINDOW Application M-file for plotwindow.fig
%    FIG = PLOTWINDOW launch plotwindow GUI.
%    PLOTWINDOW('callback_name', ...) invoke the named
(cont.)  callback.

% Last Modified by GUIDE v2.0 10-Nov-2002 14:07:54

if nargin == 0  % LAUNCH GUI

    fig = openfig(mfilename,'new');

    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundC

    % Generate a structure of handles to pass to
(cont.)  callbacks, and store it.
    handles = guihandles(fig);
    guidata(fig, handles);

    if nargout > 0
        varargout{1} = fig;
    end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION
(cont.)  OR CALLBACK

    try
        if (nargout)
            [varargout{1:nargout}] =
(cont.)  feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
    catch
        disp(lasterr);
    end
end
```

```
% --------------------------------------------------
function fig = init(parent, filename, names, points,
(cont.)  colors)
% INIT
% Create and initialize a new plot window with the
(cont.)  specified parent,
% filename, and plot data.

fig = openfig(mfilename,'new');

% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor

% Generate a structure of handles to pass to
(cont.)  callbacks
handles = guihandles(fig);

% Set the title bar to the filename
set(fig, 'Name', filename);

% Set all the handles data
handles.parent      = parent;
handles.filename    = filename;
handles.names       = names;
handles.points      = points;
handles.colors      = colors;
handles.postZoomFcn = 'plotwindow(''redraw'', gcf)';
handles.extZoomFcn  = 'plotwindow(
(cont.) (''zoomToClusterCallback'', gcf)';
guidata(fig, handles);

% Plot the graph
redraw(fig);

% --------------------------------------------------
function [filename, names, points, colors] =
(cont.)  getInfo(h)
% GETINFO
% Request plot data from this plot window.
```

```
handles     = guidata(h);
filename    = handles.filename;
names       = handles.names;
points      = handles.points;
colors      = handles.colors;

% ------------------------------------------------------
function recolor(h, names, colors)
% RECOLOR
% Change points with the given names to the specified
(cont.)  colors.
% List of point names may be a subset or superset of
(cont.)  points used
% in this plot, and may appear in an alternate order.

handles = guidata(h);

if (get(handles.holdbutton, 'Value') == 0),
    baseColors = zeros(length(handles.names),1);

    % slowcolor is O(n^2), but more robust.
    handles.colors = slowcolor(handles.names, names,
(cont.)  colors, baseColors);

    % fastcolor can work at O(n), but will fail if the
(cont.)  points are ordered
    % differently.
    % handles.colors = fastcolor(handles.names, names,
(cont.)  colors, baseColors);
    guidata(h, handles);
    redraw(h);
end

% ------------------------------------------------------
function redraw(h)
% REDRAW
% Update the plot in this window

handles = guidata(h);
```

```
axes(handles.axes1);
cla;

colorStrs = ['k','b','g','r','c','m','y'];
numColors = length(colorStrs);
symbolStrs = ['.', 'o','x','+','*','s','d','v','^','<',
(cont.) ,'>','p','h'];
numSymbols = length(symbolStrs);

colors = handles.colors;

[col, sym] = decodecolor(colors);

% Plot each color/symbol set
for s = 1:numSymbols
    symPoints = find(sym==s);
    for c = 1:numColors
        hold on;
        pointSet = intersect(find(col==c),
(cont.)  symPoints);
        plot(handles.points(pointSet,1),
(cont.)  handles.points(pointSet,2), ...
            [colorStrs(c), symbolStrs(s)], 'HitTest',
(cont.)  'off');
    end
end

% Count visible points
xl = xlim;
yl = ylim;

aboveMin = intersect(find(handles.points(:,1) >
(cont.)  xl(1)), ...
    find(handles.points(:,2) > yl(1)));
belowMax = intersect(find(handles.points(:,1) <
(cont.)  xl(2)), ...
    find(handles.points(:,2) < yl(2)));
visible = intersect(aboveMin, belowMax);

% Draw point labels if sufficiently few points are
```

```
(cont.)  visible
if (length(visible) < 50)
    text(handles.points(visible,1),
(cont.)  handles.points(visible,2), ...
        handles.names(visible));
end


% ------------------------------------------------------
function varargout = selectPointsFcn(h)
% SELECTPOINTSFCN
% Called when in select points mode, and the user
(cont.)  clicks on the window
% to recolor a portion of the graph.

handles = guidata(h);

point1 = get(handles.axes1,'CurrentPoint');    %
(cont.)  starting location
finalRect = rbbox;                             % wait
(cont.)  for user to drag box
point2 = get(handles.axes1,'CurrentPoint');    %
(cont.)  ending location
point1 = point1(1,1:2);                        %
(cont.)  extract x and y
point2 = point2(1,1:2);
minPoint = min(point1,point2);                 %
(cont.)  calculate min, max
maxPoint = max(point1,point2);

% We're now the active plot
controller('setActivePlot', handles.parent, h);

% Get the selected region
aboveMin = intersect(find(minPoint(1) <
(cont.)  handles.points(:,1)), ...
    find(minPoint(2) < handles.points(:,2)));
belowMax = intersect(find(maxPoint(1) >
(cont.)  handles.points(:,1)), ...
    find(maxPoint(2) > handles.points(:,2)));
```

```
selectedPoints = intersect(aboveMin, belowMax);

% Don’t bother doing any recoloring if there are no
(cont.)  points
if (length(selectedPoints) == 0),
    return;
end

% If colors are not being held, recolor all plots.
% Otherwise, just recolor this one.
if (get(handles.holdbutton, ’Value’) == 0),
    controller(’plotPointSelectionCallback’,
(cont.)  handles.parent, ...
        handles.names(selectedPoints));
else,
    selectInternal(h, handles.names(selectedPoints),
(cont.)  ...
        controller(’getSelectColor’,
(cont.)  handles.parent));
end


% ----------------------------------------------------
function zoomToClusterCallback(h)
% ZOOMTOCLUSTERCALLBACK
% Called on a shift-click or simultaneous click of
(cont.)  right and left mouse
% buttons, when using the zoom command.

handles = guidata(h);
clickPoint = get(handles.axes1,’CurrentPoint’);    %
(cont.)  starting location
basex = clickPoint(1,1);
basey = clickPoint(1,2);

% Get the selected region
xdiff = basex - handles.points(:,1);
ydiff = basey - handles.points(:,2);
dist  = sqrt(xdiff.*xdiff + ydiff.*ydiff);
[mindist,closest] = min(dist);
```

```
controller('zoomToCluster', handles.parent,
(cont.)  handles.colors(closest));

% ------------------------------------------------------
function select(h, names, color)
% SELECT
% Request a recoloring of the given names to be the
(cont.)  given color.
% Under certain circumstances (i.e. the 'Hold Colors'
(cont.)  button is
% toggled on), the plotwindow may not honor this
(cont.)  request.

handles = guidata(h);

if (get(handles.holdbutton, 'Value') == 0),
    selectInternal(h, names, color);
end

% ------------------------------------------------------
function selectInternal(h, names, color)
% SELECTINTERNAL
% Recolors the specified names to be the given color
(cont.)  and redraws the
% plot.  Does the actual work of selecting; should
(cont.)  only be called
% internally.  Outside parties should use SELECT.

handles = guidata(h);
handles.colors(ismember(handles.names, names)) =
(cont.)  color;
guidata(h, handles);
redraw(h);

% ------------------------------------------------------
function clearColors(h)
% CLEARCOLORS
% Request that all points be recolored as black dots.
(cont.)  May not be
```

```
% completed if 'Hold Colors' toggle button is
(cont.)  depressed.

handles = guidata(h);

if (get(handles.holdbutton, 'Value') == 0),
    handles.colors = zeros(length(handles.colors),1);
    guidata(h, handles);
    redraw(h);
end

% ------------------------------------------------------
function zoomToCluster(h, color)
% ZOOMTOCLUSTER
% Change axis scaling to include all points of
(cont.)  specified color

handles = guidata(h);

clusterPoints = handles.points(find(handles.colors ==
(cont.)  color),:);
if (length(clusterPoints) == 0),
    return;
end

mins = min(clusterPoints,[],1);
maxs = max(clusterPoints,[],1);

set(handles.axes1, 'xlim', [mins(1), maxs(1)]);
set(handles.axes1, 'ylim', [mins(2), maxs(2)]);
redraw(h);

% ------------------------------------------------------
function preZoomFcn(h)
% PREZOOMFCN
% Called in place of zoom when in zoom mode.
(cont.)  Functions as a wrapper,
% so that we can record the active plot before doing
(cont.)  the actual zooming.
```

```
handles = guidata(h);

% Tell our parent that we’re now the active window
controller(’setActivePlot’, handles.parent, h);

% Do the actual zoom, as well.
zoom(h, ’down’);

% ----------------------------------------------------
function varargout = figure1_DeleteFcn(h, eventdata,
(cont.)  handles, varargin)
% FIGURE1_DELETEFCN
% Automatically called when plot is closed.  Notifies
(cont.)  parent controller,
% so it can keep track of which plots are still open.

% Notify our parent that we no longer exist
controller(’plotClosingCallback’, handles.parent, h);


% ----------------------------------------------------
function varargout = figure1_ResizeFcn(h, eventdata,
(cont.)  handles, varargin)
% FIGURE1_RESIZEFCN
% Automatically called when plot window is resized.

% TODO -- eventually someone can write this routine to
(cont.)  do the layout
% manually, to make the window look nicer.

% ----------------------------------------------------
function enableZoom(h)
% ENABLEZOOM
% Switch this plot into zoom mode.

handles = guidata(h);
zoom(h, ’ON’);
set(h, ’WindowButtonDownFcn’,
(cont.)  ’plotwindow(’’preZoomFcn’’, gcbo)’);
set(handles.text1, ’String’, ...
    strvcat('Left-click to zoom in; Right-click to
(cont.)  zoom out', ...
     'Shift-click to zoom to cluster; Double-click to
(cont.)  reset zoom'));

% ----------------------------------------------------
function disableZoom(h)
% DISABLEZOOM
% Switch this plot out of zoom mode.

handles = guidata(h);
set(handles.text1, 'String', '');
zoom(h, 'OFF');

% ----------------------------------------------------
function enableSelect(h)
% ENABLESELECT
% Switch this plot into select mode.

handles = guidata(h);
set(h, 'WindowButtonDownFcn',
(cont.)  'plotwindow(''selectPointsFcn'', gcbo)');
set(handles.text1, 'String', 'Click and drag to
(cont.)  recolor points.');

% ----------------------------------------------------
function disableSelect(h)
% DISABLESELECT
% Switch this plot out of select mode.

handles = guidata(h);
set(handles.text1, 'String', '');
set(h, 'WindowButtonDownFcn', []);
```

B.19 randDaviesBouldin.m

```
%Runs 100 iterations of the DB index with random
(cont.)  labelings, then
%shows where the DB value for the input labeling falls
(cont.)  on the
%probability density function shown.  This one’s
(cont.)  pretty quick.

function [output, retfig] = randDaviesBouldin(
(cont.) (goodLabels, points)

numClusters = max(goodLabels);
numPoints = length(goodLabels);

iterations = 100;
DBValues = zeros(iterations, 1);

for iter = 1:iterations

    labels = ceil(rand(size(goodLabels)) *
(cont.)  numClusters);
    DBValues(iter) = DaviesBouldin(labels, points);

end

actualDB = DaviesBouldin(goodLabels, points);

retfig = figure;
hist(DBValues, 20);
hold on
title(’Probability distribution for random labels:
(cont.)  Davies-Bouldin’);
xlabel(’Davies-Bouldin’);
stem(actualDB, iterations / 10, ’r:’);

output = sprintf(’Davies-Bouldin = %d’, actualDB);
```

B.20 randMHG.m

```
%Monte Carlo testing using the random label hypothesis
(cont.)  and
%the modified Hubert Gamma statistic.  This takes a
(cont.)  long
%time.  For data sets with more than 3000 or so
(cont.)  points,
%100 iterations may take 20+ minutes.

function [output, retfig] = randMHG(goodLabels,
(cont.)  points)

numClusters = max(goodLabels);
numPoints = length(goodLabels);

iterations = 100;
HGValues = zeros(iterations, 2);

for iter = 1:iterations

    labels = ceil(rand(size(goodLabels)) *
(cont.)  numClusters) + 1;
    values = modifiedHubertGamma(labels, points);
    HGValues(iter, 1) = values(1);
    HGValues(iter, 2) = values(2);

end

goodValues = modifiedHubertGamma(goodLabels, points);
actualHubert = goodValues(1);
actualNormHubert = goodValues(2);

retfig = figure;
subplot(2, 1, 1);
hist(HGValues(:, 1), 20);
hold on
xlabel('Modified Hubert Gamma');
title('Probability distribution for random labels:
(cont.)  Modified Hubert Gamma');
```

```
stem(actualHubert, iterations / 2, ’r:’);
subplot(2, 1, 2);
hist(HGValues(:, 2), 20);
hold on
xlabel(’Normalized Hubert Gamma’);
title(’Probability distribution for random labels:
(cont.)  Normalized modified Hubert Gamma’);
stem(actualNormHubert, iterations / 2, ’r:’);

output = sprintf(’Modified Hubert Gamma = %d’,
(cont.)  actualHubert);
output = strvcat(output, sprintf(’Normalized Modified
(cont.)  Hubert Gamma = %d\n’, ...
                                  actualNormHubert));
```

B.21 randRandTypes.m

```
%Monte Carlo testing using Rand type statistics.  This
(cont.)  requires two
%good labelings of the data points to be compared.

function [output, retfig] = randRandTypes(goodLabels1,
(cont.)  points1, goodLabels2, points2)

iterations = 100;

if nargin == 2
    maxLabel = length(unique(goodLabels1));
    goodLabels2 = ceil(rand(size(goodLabels1)) *
(cont.)  maxLabel);
    points2 = points1;
end

M = length(goodLabels1) * (length(goodLabels1) - 1) /
(cont.)  2;
maxLabel = length(unique(goodLabels1));
metricDist = zeros(4, iterations);

for iter = 1:iterations

    labels2 = ceil(rand(size(goodLabels1)) *
(cont.)  maxLabel);
    randValues = randTypes(goodLabels1, points1,
(cont.)  labels2, points2);
    metricDist(1, iter) = randValues(1);
    metricDist(2, iter) = randValues(2);
    metricDist(3, iter) = randValues(3);
    metricDist(4, iter) = randValues(4);

end

randValues = randTypes(goodLabels1, points1,
(cont.)  goodLabels2, points2);

retfig = figure;
```

```
subplot(4, 1, 1);
hist(metricDist(1, :), 20);
hold on
xlabel('Rand');
title('Probability distribution for random labels:
(cont.)  Rand statistic');
stem(randValues(1), iterations / 4, 'r:');
subplot(4, 1, 2);
hist(metricDist(2, :), 20);
hold on
xlabel('Jaccard');
title('Probability distribution for random labels:
(cont.)  Jaccard coefficient');
stem(randValues(2), iterations / 4, 'r:');
subplot(4, 1, 3);
hist(metricDist(3, :), 20);
hold on
xlabel('FM');
title('Probability distribution for random labels:
(cont.)  Fowlkes-Mallows Index');
stem(randValues(3), iterations / 4, 'r:');
subplot(4, 1, 4);
hist(metricDist(4, :), 20);
hold on
xlabel('Hubert');
title('Probability distribution for random labels:
(cont.)  Hubert Gamma');
stem(randValues(4), iterations / 4, 'r:');

output = sprintf('Rand statistic: %f',
(cont.)  randValues(1));
output = strvcat(output, sprintf('Jaccard coefficient:
(cont.)  %f', randValues(2)));
output = strvcat(output, sprintf('Fowlkes & Mallows
(cont.)  index: %f', ...
    randValues(3)));
output = strvcat(output, sprintf('Hubert Gamma: %f',
(cont.)  randValues(4)));
```

B.22 randTypes.m

```
function output = randTypes(labels1, points1, labels2,
(cont.)  points2)

% Compares two colorings to generate the Rand
(cont.)  statistic, Jaccard coefficient,
% Fowlkes & Mallows index, and Hubert statistic.

ss = 0;
sd = 0;
ds = 0;
dd = 0;

for i = 1:length(labels1) - 1
    for j = i + 1:length(labels1)
        if (labels1(i) == labels1(j)) & (labels2(i) ==
(cont.)  labels2(j))
            ss = ss + 1;
        elseif (labels1(i) ~= (labels1(j)) &
(cont.)  (labels2(i) ~= labels2(j)))
            dd = dd + 1;
        elseif (labels1(i) ~= (labels1(j)) &
(cont.)  (labels2(i) == labels2(j)))
            ds = ds + 1;
        elseif (labels1(i) == (labels1(j)) &
(cont.)  (labels2(i) ~= labels2(j)))
            sd = sd + 1;
        end
    end
end

M = length(labels1) * (length(labels1) - 1) / 2;
m1 = ss + sd;
m2 = ss + ds;

output = zeros(4, 1);
```

```
output(1) = (ss + dd) / M;
output(2) = ss / (ss + sd + ds);
output(3) = sqrt((ss / m1) * (ss / m2));
output(4) = (M * ss - m1 * m2) / sqrt(m1 * m2 * (M -
(cont.)  m1) * (M - m2));
```

B.23 randTypesWrapper.m

```
function [output, retfig] = randTypesWrapper(labels1,
(cont.)  points1, labels2, points2)

% Compares two colorings to generate the Rand
(cont.)  statistic, Jaccard coefficient,
% Fowlkes & Mallows index, and Hubert statistic.
(cont.)  Returns a description of these
% as a string.

if nargin == 2
    maxLabel = length(unique(labels1));
    labels2 = ceil(rand(size(labels1)) * maxLabel);
    points2 = points1;
end

randValues = randTypes(labels1, points1, labels2,
(cont.)  points2);

output = sprintf('Rand statistic: %f',
(cont.)  randValues(1));
output = strvcat(output, sprintf('Jaccard coefficient:
(cont.)  %f', randValues(2)));
output = strvcat(output, sprintf('Fowlkes & Mallows
(cont.)  index: %f', ...
                                randValues(3)));
output = strvcat(output, sprintf('Hubert Gamma: %f',
(cont.)  randValues(4)));

% No new figures
retfig = [];
```

B.24 randomLabelHyp.m

```
function output = randomLabelHyp(goodLabels, points)

iterations = 200;

M = length(goodLabels) * (length(goodLabels) - 1) /
(cont.)  2;

metricDist = zeros(3, iterations);

for iter = 1:iterations

    ss = 0;
    sd = 0;
    ds = 0;
    dd = 0;

    %Not right...note labels aren’t continuous yet, I
(cont.)  think
    maxLabel = length(unique(goodLabels));
    labels1 = ceil(rand(size(goodLabels)) *
(cont.)  maxLabel);
    labels2 = ceil(rand(size(goodLabels)) *
(cont.)  maxLabel);

    for i = 1:length(labels1) - 1
        for j = i + 1:length(labels1)
            if (labels1(i) == labels1(j)) &
(cont.)  (labels2(i) == labels2(j))
                ss = ss + 1;
            elseif (labels1(i) ˜= (labels1(j)) &
(cont.)  (labels2(i) ˜= labels2(j)))
                dd = dd + 1;
            elseif (labels1(i) ˜= (labels1(j)) &
(cont.)  (labels2(i) == labels2(j)))
                ds = ds + 1;
            elseif (labels1(i) == (labels1(j)) &
(cont.)  (labels2(i) ˜= labels2(j)))
                sd = sd + 1;
            end
        end
    end

    metricDist(1, iter) = (ss + dd) / M;
    metricDist(2, iter) = ss / (ss + sd + ds);
    metricDist(3, iter) = sqrt((ss / (ss + sd)) * (ss
(cont.)  / (ss + ds)));

end

%Actual stats

ss = 0;
sd = 0;
ds = 0;
dd = 0;

labels1 = goodLabels;

for i = 1:length(labels1) - 1
    for j = i + 1:length(labels1)
        if (labels1(i) == labels1(j)) & (labels2(i) ==
(cont.)  labels2(j))
            ss = ss + 1;
        elseif (labels1(i) ~= (labels1(j)) &
(cont.)  (labels2(i) ~= labels2(j)))
            dd = dd + 1;
        elseif (labels1(i) ~= (labels1(j)) &
(cont.)  (labels2(i) == labels2(j)))
            ds = ds + 1;
        elseif (labels1(i) == (labels1(j)) &
(cont.)  (labels2(i) ~= labels2(j)))
            sd = sd + 1;
        end
    end
end

actualRand = (ss + dd) / M;
actualJaccard = ss / (ss + sd + ds);
```

```
actualFM = sqrt((ss / (ss + sd)) * (ss / (ss +
(cont.)  ds)));

f = figure(89);

subplot(3, 1, 1);
h = hist(metricDist(1, :), 20);
bar(h);
hold on
xlabel('Rand');
stem(actualRand, max(h), 'r:');
subplot(3, 1, 2);
h = hist(metricDist(2, :), 20);
bar(h);
hold on
xlabel('Jaccard');
stem(actualJaccard, max(h), 'r:');
subplot(3, 1, 3);
h = hist(metricDist(3, :), 20);
bar(h);
hold on
xlabel('FM');
stem(actualFM, max(h), 'r:');

output = sprintf('Rand statistic: %f', (ss + dd) /
(cont.)  M);
output = strvcat(output, sprintf('Jaccard coefficient:
(cont.)  %f', ...
    ss / (ss + sd + ds)));
output = strvcat(output, sprintf('Fowlkes & Mallows
(cont.)  index: %f', ...
    sqrt((ss / (ss + sd)) * (ss / (ss + ds)))));
```

B.25 savecontrol.m

```
function varargout = savecontrol(varargin)
% SAVECONTROL Application M-file for savecontrol.fig
%    FIG = SAVECONTROL launch savecontrol GUI.
%    SAVECONTROL('callback_name', ...) invoke the
(cont.)  named callback.
%
% The Save Colors dialog enables the user to save the
(cont.)  coloring of an
% open plot to a file.  Unlike most other dialogs, it
(cont.)  is modal.

% Last Modified by GUIDE v2.0 11-Feb-2003 17:46:14

if nargin == 0  % LAUNCH GUI

fig = openfig(mfilename,'reuse');

% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColo

% Generate a structure of handles to pass to
(cont.)  callbacks, and store it.
handles = guihandles(fig);
guidata(fig, handles);

if nargout > 0
varargout{1} = fig;
end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION
(cont.)  OR CALLBACK

try
[varargout{1:nargout}] = feval(varargin{:}); % FEVAL
(cont.)  switchyard
catch
disp(lasterr);
end
```

```
end

% ----------------------------------------------------
function fig = init(parent, pathname, plotnames)
% INIT
% Create a new Save Colors dialog with the given
(cont.)  default path and set of plot
% names to select from.

fig = openfig(mfilename,'reuse');

% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor

% Generate a structure of handles to pass to
(cont.)  callbacks, and store it.
handles = guihandles(fig);
handles.parent = parent;
handles.pathname = pathname;
handles.fig = fig;
guidata(fig, handles);

% Set the list of plots
set(handles.plotmenu, 'String', plotnames);


% ----------------------------------------------------
function varargout = saveasbutton_Callback(h,
(cont.)  eventdata, handles, varargin)
% SAVEASBUTTON_CALLBACK
% Automatically called when the user clicks on the
(cont.)  "Save As..." button.
% Prompts user for filename and writes the selected
(cont.)  coloring to that file.
% (Actual saving is done via a callback to the
(cont.)  controller.)

% Save workplace directory, then switch to working
(cont.)  dir
```

```
currdir = pwd;
cd(handles.pathname);

% Get filename from the user
[filename, newpathname] = ...
    uiputfile({'*.color', 'Coloring (*.color)'; '*.*',
(cont.)  'All Files (*.*)'}, ...
        'Save Coloring');

% Restore workplace directory
cd(currdir);

if (isequal(filename,0))
    return;
end

filename = strcat(filename, '.color');

% Ask parent to save colors
controller('saveColorsCallback', handles.parent,
(cont.)  get(handles.plotmenu, 'Value'), ...
    newpathname, filename);

% Close ourselves -- we're done.
close(handles.fig);

% ------------------------------------------------------
function varargout = cancelbutton_Callback(h,
(cont.)  eventdata, handles, varargin)
% CANCELBUTTON
% Automatically called when user clicks on the
(cont.)  "Cancel" button.  Closes
% the Save Colors dialog rather than doing any work.

% Close this window.  We're done.
close(handles.fig);
```

B.26 selectcontrol.m

```
function varargout = selectcontrol(varargin)
% SELECTCONTROL Application M-file for
(cont.)  selectcontrol.fig
%    FIG = SELECTCONTROL launch selectcontrol GUI.
%    SELECTCONTROL('callback_name', ...) invoke the
(cont.)  named callback.
%
% The Select dialog lets the user specify the color
(cont.)  and symbol to be
% used for manually coloring open plots.

% Last Modified by GUIDE v2.5 02-Mar-2003 14:57:23

if nargin == 0  % LAUNCH GUI

    fig = selectcontrol('init', 0);

    if nargout > 0
        varargout{1} = fig;
    end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION
(cont.)  OR CALLBACK

    try
        if (nargout)
            [varargout{1:nargout}] =
(cont.)  feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
    catch
        disp(lasterr);
    end

end

% ------------------------------------------------------
```

```
function fig = init(parent, initialcolor)
% INIT
% Create a new instance of the Select dialog.  Accepts
(cont.)  two parameters,
% the parent Controller that initiated it (and to whom
(cont.)  it will make
% its callbacks), and the initial color selected.

fig = openfig(mfilename,'reuse');

% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor

% Generate a structure of handles to pass to
(cont.)  callbacks, and store it.
handles = guihandles(fig);
handles.parent = parent;

% Restore the original color selected
[handles.color, handles.symbol] =
(cont.)  decodecolor(initialcolor);
set(handles.colorPopup, 'Value', handles.color);
set(handles.symbolPopup, 'Value', handles.symbol);

guidata(fig, handles);
notifyParent(fig, handles);

% ------------------------------------------------------
function varargout = colorPopup_Callback(h, eventdata,
(cont.)  handles, varargin)
% COLORPOPUP_CALLBACK
% Automatically called when a color is selected in the
(cont.)  popup menu.
% Notifies the parent controller of this new color.

handles.color = get(handles.colorPopup, 'Value');

guidata(h, handles);
notifyParent(h, handles);
```

```
% ------------------------------------------------------
function varargout = symbolPopup_Callback(h,
(cont.)  eventdata, handles, varargin)
% SYMBOLPOPUP_CALLBACK
% Automatically called when a symbol is selected in
(cont.)  the popup menu.
% Notifies the parent controller of this new symbol.

handles.symbol = get(handles.symbolPopup, ’Value’);
guidata(h, handles);
notifyParent(h, handles);

% ------------------------------------------------------
function varargout = notifyParent(h, handles)
% NOTIFYPARENT
% Notify the parent controller of the new color/symbol
(cont.)  combination to
% be used for manual coloring.

controller(’setSelectColor’, handles.parent, ...
    encodecolor(handles.color, handles.symbol));

% ------------------------------------------------------
function varargout = figure1_DeleteFcn(h, eventdata,
(cont.)  handles, varargin)
% FIGURE1_DELETEFCN
% Called automatically when the select dialog is
(cont.)  closed.  Notifies parent
% that we no longer exist, so it can switch out of
(cont.)  select mode and stop
% worrying about us.

controller(’selectClosingCallback’, handles.parent);


% ------------------------------------------------------
function next_color_button_Callback(h, eventdata,
(cont.)  handles)
% NEXT_COLOR_BUTTON_CALLBACK
% Automatically called when "Next Color" button is
```

```
(cont.)  clicked.  Increments
% the current selection color.  If the current color
(cont.)  is the maximum color,
% the current color is reset and the symbol is
(cont.)  incremented instead.

if (handles.color < [7.0])
    handles.color= handles.color+1.0;
elseif (handles.symbol < [13.0])
    handles.color=[1.0];
    handles.symbol = handles.symbol+1.0;
else
    handles.color=[1.0];
    handles.symbol=[1.0];
end

set(handles.colorPopup,’Value’,handles.color);
set(handles.symbolPopup,’Value’,handles.symbol);
guidata(h, handles);
notifyParent(h, handles);
```

B.27 slowcolor.m

```
function colors2 = slowcolor(names2, names1, colors1,
(cont.)  baseColors)
% SLOWCOLOR
% Given a mapping of names to colors, assign colors to
(cont.)  another set of names.
% O(nˆ2), but guaranteed to work.

% Start out with the base, default colors
colors2 = baseColors;

% Iterate through common points
common = find(ismember(names1, names2))’;
for i = common,

    % Find the matching index
    j = find(strcmpi(names2, names1(i)));

    % Set the color
    colors2(j) = colors1(i);
end
```

B.28 subDaviesBouldin.m

```
function output = subDaviesBouldin(labels, points)

numClusters = max(labels);
numPoints = length(labels);

%Find the center of each cluster

clusterPoints = ones(numClusters, 1);
xcenters = zeros(numClusters, 1);
ycenters = zeros(numClusters, 1);

for i = 1:numPoints
    current = labels(i);
    clusterPoints(current) = clusterPoints(current) +
(cont.)  1;
    xcenters(current) = xcenters(current) + points(i,
(cont.)  1);
    ycenters(current) = ycenters(current) + points(i,
(cont.)  2);
end

xcenters = xcenters ./ clusterPoints;
ycenters = ycenters ./ clusterPoints;

actualClusters = numClusters;
errors = zeros(numClusters, 1);
for i = 1:numClusters
    c = find(labels == i);
    if length(c) == 0
        actualClusters = actualClusters - 1;
    else
        for j = 1:length(c)
            c(j) = sqrt((points(c(j), 1) -
(cont.)  xcenters(i))^2 + ...
                        (points(c(j), 2) -
(cont.)  ycenters(i))^2);
        end
        errors(i) = var(c);
    end
end

DB = 0;

for i = 1:numClusters
    max = 0;
    for j = 1:numClusters
        Rij = 0;
        denom = sqrt((xcenters(i) - xcenters(j))ˆ2 +
(cont.)  ...
                     (ycenters(i) - ycenters(j))ˆ2);
(cont.)
        if denom ˜= 0
            Rij = (errors(i) + errors(j)) / denom;
(cont.)
        end
        if Rij > max
            max = Rij;
        end
    end
    DB = DB + max;
end

output = DB / actualClusters;
```

B.29 submodifiedHubertGamma.m

```
function output = submodifiedHubertGamma(labels,
(cont.)  points)

numClusters = max(labels);
numPoints = length(labels);
clusterPoints = ones(numClusters, 1);
xcenters = zeros(numClusters, 1);
ycenters = zeros(numClusters, 1);

%Find the center of each cluster

for i = 1:length(labels)
    current = labels(i);
    clusterPoints(current) = clusterPoints(current) +
(cont.)  1;
    xcenters(current) = xcenters(current) + points(i,
(cont.)  1);
    ycenters(current) = ycenters(current) + points(i,
(cont.)  2);
end

xcenters = xcenters ./ clusterPoints;
ycenters = ycenters ./ clusterPoints;

%P is the proximity matrix

P = squareform(pdist(points));      %distances
P = max(max(P)) - P;                %similarities

%QQ(i, j) is the distance between cluster i and j

QQ = zeros(numClusters);
for i = 1:numClusters
    for j = 1:numClusters
        QQ(i, j) = sqrt((xcenters(i) - xcenters(j))^2
(cont.)  + ...
                       (ycenters(i) -
(cont.)  ycenters(j))^2);
```

```
    end
end

%Q(i, j) is the distance between the center of the
(cont.)  cluster to which
%point i belongs and the center of the cluster to
(cont.)  which point j
%belongs.

Q = zeros(size(P));

for i = 1:numPoints - 1
    for j = i + 1:numPoints
        Q(i, j) = QQ(labels(i), labels(j));
    end
end

hubert = 0;
normHubert = 0;

%Find the mean of the upper right diagonal of P and
(cont.)  Q

sumP = 0;
sumQ = 0;
for i = 1:numPoints - 1
    for j = i + 1:numPoints
        sumP = sumP + P(i, j);
        sumQ = sumQ + Q(i, j);
    end
end

meanP = sumP / (numPoints * (numPoints - 1) / 2);
meanQ = sumQ / (numPoints * (numPoints - 1) / 2);
meanPsq = meanPˆ2;
meanQsq = meanQˆ2;

%Find the variance of the upper right diagonal of P
(cont.)  and Q
```

```
stdevP = 0;
stdevQ = 0;
for i = 1:numPoints - 1
    for j = i + 1:numPoints
        stdevP = stdevP + P(i, j)^2 - meanPsq;
        stdevQ = stdevQ + Q(i, j)^2 - meanQsq;
    end
end

stdevP = stdevP / (numPoints * (numPoints - 1) / 2);
stdevQ = stdevQ / (numPoints * (numPoints - 1) / 2);
varP = sqrt(stdevP);
varQ = sqrt(stdevQ);

for i = 1:numPoints - 1
    for j = i + 1:numPoints
        hubert = hubert + P(i, j) * Q(i, j);
        normHubert = normHubert + (P(i, j) - meanP) *
(cont.)  ...
                                  (Q(i, j) - meanQ);
    end
end



hubert = hubert / (numPoints * (numPoints - 1) / 2);
normHubert = normHubert / (numPoints * (numPoints - 1)
(cont.)  / 2);

normHubert = normHubert / (varP * varQ);

output = [hubert, normHubert];
%output = sprintf('Modified Hubert Gamma = %d',
(cont.)  hubert);
%output = strvcat(output, sprintf('Normalized Modified
(cont.)  Hubert Gamma = %d\n', ...
%                                  normHubert));
```

B.30 validitycontrol.m

```
function varargout = validitycontrol(varargin)
% COMPARECONTROL Application M-file for
(cont.)  validitycontrol.fig
%    FIG = COMPARECONTROL launch validitycontrol GUI.
%    COMPARECONTROL('callback_name', ...) invoke the
(cont.)  named callback.
%
% The Validity dialog box enables the user to run
(cont.)  various metrics using the
% points and clusterings of open plots, and displays
(cont.)  the results in a
% text box.

% Last Modified by GUIDE v2.0 15-Mar-2003 14:45:04

if nargin == 0  % LAUNCH GUI

    % We should never reach this point.  See INIT.

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION
(cont.)  OR CALLBACK

try
[varargout{1:nargout}] = feval(varargin{:}); % FEVAL
(cont.)  switchyard
catch
disp(lasterr);
end

end

% ----------------------------------------------------
function fig = init(parent)
% INIT
% Create a new instance of the Validity GUI.  Accepts
(cont.)  one parameters,
% the parent Controller that initiated it (and to whom
(cont.)  it will make
```

```
% its callbacks).

fig = openfig(mfilename,'reuse');

% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor

% Generate a structure of handles to pass to
(cont.)  callbacks, and store it.
handles = guihandles(fig);
handles.parent = parent;
handles.metric = 1;         % Validity metric and
(cont.)  sub-metric
handles.option = 1;
handles.table = metrictable;

allEntries = [handles.table{:}];
set(handles.metricpopup, 'String',
(cont.)  strvcat(allEntries.name));
guidata(fig, handles);
updateUI(fig);


% ------------------------------------------------------
function varargout = updatePlotNames(h, names)
% UPDATEPLOTNAMES
% Update popup menus to show a new list of available
(cont.)  plots on which
% to run metrics.  If no plots are available, disable
(cont.)  the "Run Metric"
% button as well.

handles = guidata(h);

if (length(names) == 0),
    % If there are no plots, disable everything.
    set(handles.primarypopup, 'String', '<none>');
    set(handles.secondarypopup, 'String', 'Random');
    set(handles.runbutton, 'Enable', 'off');
else,
    % Update popup menus to show possible plots to run
(cont.)  metrics on.
    set(handles.primarypopup, ’String’, names);
    set(handles.secondarypopup, ’String’,
(cont.)  strvcat(’Random’, names));
    set(handles.runbutton, ’Enable’, ’on’);
end

% ------------------------------------------------------
function varargout = runbutton_Callback(h, eventdata,
(cont.)  handles, varargin)
% RUNBUTTON_CALLBACK
% Automatically called when the user clicks the "Run
(cont.)  Metric" button.
% Gets the metric information from the metric table,
(cont.)  asks the parent
% controller to run the metric, and displays the
(cont.)  results in the text box.

index1 = get(handles.primarypopup, ’Value’);

if (strcmpi(get(handles.secondarypopup, ’Enable’),
(cont.)  ’on’)),
    index2 = get(handles.secondarypopup, ’Value’) -
(cont.)  1;
else,
    index2 = 0;
end

metricname = handles.table{handles.metric}.name;
optionname = handles.table{handles.metric}.options{han
numargs    = handles.table{handles.metric}.options{han
method     = handles.table{handles.metric}.options{han

% Compute and display results

results = [’Metric: ’, metricname];

if (strcmpi(get(handles.optionpopup, ’Enable’),
(cont.)  ’on’)),
    results = strvcat(results, ['Option: ',
(cont.)  optionname]);
end

plotnames = get(handles.primarypopup, 'String');
results = strvcat(results, ['Primary Plot: ',
(cont.)  plotnames(index1, :)]);

if (index2 > 0),
    results = strvcat(results, ['Secondary Plot: ',
(cont.)  plotnames(index2, :)]);
end

% Actual metric results are obtained and appended
(cont.)  here
results = strvcat(results, controller(
(cont.) ('validityCallback', handles.parent, ...
    method, index1, index2));

set(handles.resultsbox, 'String', results);

% ----------------------------------------------------
function varargout = figure1_DeleteFcn(h, eventdata,
(cont.)  handles, varargin)
% FIGURE1_DELETEFCN
% Automatically called when the validity dialog is
(cont.)  closed.  Notifies
% the

controller('validityClosingCallback',
(cont.)  handles.parent);


% ----------------------------------------------------
function varargout = metricpopup_Callback(h,
(cont.)  eventdata, handles, varargin)
% METRICPOPUP_CALLBACK
% Automatically called when a new validity metric is
(cont.)  selected.  Changes the
% active metric, resets the metric option to the
```

```
(cont.)  default, then updates the
% rest of the UI to reflect this change.

metric = get(handles.metricpopup, ’Value’);

if (metric ˜= handles.metric),
    handles.metric = metric;
    handles.option = 1;
    guidata(h, handles);
    updateUI(h);
end

% ----------------------------------------------------
function varargout = optionpopup_Callback(h,
(cont.)  eventdata, handles, varargin)
% OPTIONPOPUP_CALLBACK
% Automatically called when a new validity metric
(cont.)  option is selected.
% Updates the rest of the UI to reflect this change.

newoption = get(handles.optionpopup, ’Value’);
if (newoption ˜= handles.option),
    handles.option = newoption;
    guidata(h, handles);
    updateUI(h);
end

% ----------------------------------------------------
function updateUI(h)
% UPDATEUI
% Fix up all the popup menus, enable and disable
(cont.)  buttons appropriately,
% etc. to respond to a change in validity metric or
(cont.)  option.

handles = guidata(h);

% Make sure popups are up-to-date
set(handles.metricpopup, ’Value’, handles.metric);
set(handles.optionpopup, ’String’, ...
    strvcat(handles.table{handles.metric}.options{:,
(cont.)  1}));
set(handles.optionpopup, ’Value’, handles.option);

% Enable/Disable option selection
[numoptions, optionsize] = size(
(cont.) (handles.table{handles.metric}.options);
if (numoptions == 1),
    set(handles.optionpopup, ’Enable’, ’off’);
else
    set(handles.optionpopup, ’Enable’, ’on’);
end

% Enable/Disable choice of second plot
numargs = handles.table{handles.metric}.options{handle
if (numargs == 2),
    set(handles.secondarypopup, ’Enable’, ’on’);
else,
    set(handles.secondarypopup, ’Enable’, ’off’);
end
```

B.31 zoom.m

```
function out = zoom(varargin)
%ZOOM   Zoom in and out on a 2-D plot.
%   ZOOM with no arguments toggles the zoom state.
%   ZOOM(FACTOR) zooms the current axis by FACTOR.
%       Note that this does not affect the zoom
(cont.)  state.
%   ZOOM ON turns zoom on for the current figure.
%   ZOOM OFF turns zoom off in the current figure.
%   ZOOM OUT returns the plot to its initial (full)
(cont.)  zoom.
%   ZOOM XON or ZOOM YON turns zoom on for the x or y
(cont.)  axis only.
%   ZOOM RESET clears the zoom out point.
%
%   When zoom is on, click the left mouse button to
(cont.)  zoom in on the
%   point under the mouse.  Click the right mouse
(cont.)  button to zoom out.
%   Each time you click, the axes limits will be
(cont.)  changed by a factor
%   of 2 (in or out).  You can also click and drag to
(cont.)  zoom into an area.
%   Double clicking zooms out to the point at which
(cont.)  zoom was first
%   turned on for this figure.  Note that turning zoom
(cont.)  on, then off
%   does not reset the zoom point.  This may be done
(cont.)  explicitly with
%   ZOOM RESET.
%
%   ZOOM(FIG,OPTION) applies the zoom command to the
(cont.)  figure specified
%   by FIG. OPTION can be any of the above arguments.

%   ZOOM FILL scales a plot such that it is as big as
(cont.)  possible
%   within the axis position rectangle for any azimuth
(cont.)  and elevation.
```

```
%   Clay M. Thompson 1-25-93
%   Revised 11 Jan 94 by Steven L. Eddins
%   Copyright 1984-2000 The MathWorks, Inc.
%   $Revision: 1.4 $  $Date: 2003/05/09 08:48:10 $

%   Note: zoom uses the figure buttondown and
(cont.)  buttonmotion functions
%
%   ZOOM XON zooms x-axis only
%   ZOOM YON zooms y-axis only

switch nargin,

   %%%%%%%%%%%%%%%%%%%%%%%%%%
   %%% No Input Arguments %%%
   %%%%%%%%%%%%%%%%%%%%%%%%%%
case 0,
 fig=get(0,'currentfigure');
 if isempty(fig), return, end
 zoomCommand='toggle';

   %%%%%%%%%%%%%%%%%%%%%%%%%%
   %%% One Input Argument %%%
   %%%%%%%%%%%%%%%%%%%%%%%%%%
case 1,

   % If the argument is a string, the argument is a
(cont.)  zoom command
   % (i.e. (on, off, down, xdown, etc.).  Otherwise,
(cont.)  the argument is
   % assumed to be a zoom factor.

   if isstr(varargin{1}),
       fig=get(0,'currentfigure');
       if isempty(fig), return, end

       zoomCommand=varargin{1};
   else
      scale_factor=varargin{1};
      zoomCommand='scale';
      fig=get(0,'currentfigure');
      if isempty(fig), return, end
   end % if

   %%%%%%%%%%%%%%%%%%%%%%%%%%%
   %%% Two Input Arguments %%%
   %%%%%%%%%%%%%%%%%%%%%%%%%%%
case 2,
   fig=varargin{1};
   if ~ishandle(fig), error('First argument must be a
(cont.)  figure handle.'), end
   zoomCommand=varargin{2};

otherwise,
   error(nargchk(0, 2, nargin));

end % switch nargin

zoomCommand=lower(zoomCommand);

%
% handle 'off' commands first
%
if strcmp(zoomCommand,'off'),
   %
   % turn off zoom, and take a hike
   %
   set(findall(fig,'Tag','figToolZoomIn'),'State',
(cont.) ,'off');
   set(findall(fig,'Tag','figToolZoomOut'),'State',
(cont.) ,'off');
   if ~isempty(getappdata(fig,'ZoomFigureMode'))
      rmappdata(fig,'ZOOMFigureMode');
   end

   state = getappdata(fig,'ZOOMFigureState');
   if ~isempty(state),
       % since we don't set the pointer, make sure it
(cont.)  does not get reset
```

```
       ptr = get(fig,'pointer');
       uirestore(state,'nochildren');
       set(fig,'pointer',ptr)
       rmappdata(fig,'ZOOMFigureState');
   end
   return
end % if

ax=get(fig,'currentaxes');

rbbox_mode = 0;
zoomx = 1; zoomy = 1; % Assume no constraints

if ~isempty(ax) & any(get(ax,'view')~=[0 90]) ...
           & ~(strcmp(zoomCommand,'scale') | ...
           strcmp(zoomCommand,'fill')),
   fZoom3d = 1;
   % set(findall(fig,'Tag','figToolZoom'),'State',
(cont.) ,'off');
   % warning('Zoom is only supported for 2D plots.');
   % return % Do nothing
else
   fZoom3d = 0;
end

if strcmp(zoomCommand,'toggle'),
   state = getappdata(fig,'ZOOMFigureState');
   if isempty(state)
      zoom(fig,'on');
   else
      zoom(fig,'off');
   end
   return
end % if

% Catch constrained zoom
if strcmp(zoomCommand,'xdown'),
   zoomy = 0; zoomCommand = 'down'; % Constrain y
elseif strcmp(zoomCommand,'ydown')
   zoomx = 0; zoomCommand = 'down'; % Constrain x
```

```
end

if (nargout ~= 0) & ...
   ~isequal(zoomCommand,'getmode') & ...
   ~isequal(zoomCommand,'getlimits') & ...
   ~isequal(zoomCommand,'getconnect')
   error(['ZOOM only returns an output if the command
(cont.)  is getmode,' ...
  ' getlimits, or getconnect']);
end

switch zoomCommand
 case 'down',
   % Activate axis that is clicked in
   allAxes = findall(datachildren(fig),'flat','type',
(cont.) ,'axes');
   ZOOM_found = 0;

   % this test may be causing failures for 3d axes
   for i=1:length(allAxes),
      ax=allAxes(i);
      ZOOM_Pt1 = get(ax,'CurrentPoint');
      xlim = get(ax,'xlim');
      ylim = get(ax,'ylim');
      if (xlim(1) <= ZOOM_Pt1(1,1) & ZOOM_Pt1(1,1) <=
(cont.)  xlim(2) & ...
            ylim(1) <= ZOOM_Pt1(1,2) & ZOOM_Pt1(1,2)
(cont.)  <= ylim(2))
         ZOOM_found = 1;
         set(fig,'currentaxes',ax);
         break
      end % if
   end % for

   if ZOOM_found==0, return, end

   % Check for selection type
   selection_type = get(fig,'SelectionType');
   zoomMode = getappdata(fig,'ZOOMFigureMode');
   axz = get(ax,’ZLabel’);

      if fZoom3d

      viewData = getappdata(axz,’ZOOMAxesView’);
      if isempty(viewData)
         viewProps = { ’CameraTarget’...
                    ’CameraTargetMode’...
                    ’CameraViewAngle’...
                    ’CameraViewAngleMode’};
         setappdata(axz,’ZOOMAxesViewProps’,
(cont.)  viewProps);
         setappdata(axz,’ZOOMAxesView’,
(cont.)  get(ax,viewProps));
      end

      if isempty(zoomMode) | strcmp(zoomMode,’in’);
         zoomLeftFactor = 1.5;
         zoomRightFactor = .75;
      elseif strcmp(zoomMode,’out’);
         zoomLeftFactor = .75;
         zoomRightFactor = 1.5;
      end

      switch selection_type
         case ’open’
            set(ax,getappdata(axz,’ZOOMAxesViewProps’),
(cont.) ,...
                    getappdata(axz,’ZOOMAxesView’));
         case ’normal’
            newTarget = mean(get(ax,’CurrentPoint’),
(cont.) ,1);
            set(ax,’CameraTarget’,newTarget);
            camzoom(ax,zoomLeftFactor);
         otherwise
            newTarget = mean(get(ax,’CurrentPoint’),
(cont.) ,1);
            set(ax,’CameraTarget’,newTarget);
            camzoom(ax,zoomRightFactor);
      end
```

```
         return
      end

      if isempty(zoomMode) | strcmp(zoomMode,'in');
         switch selection_type
            case 'normal'
               % Zoom in
               m = 1;
               scale_factor = 2; % the default zooming
(cont.)     factor
            case 'open'
               % Zoom all the way out
               zoom(fig,'out');
               return;
            case 'extend'
                % Call user-specified callback to do some
(cont.)     alternate zoom method.
                handles = guidata(fig);
                extZoomFcn = handles.extZoomFcn;
                if (extZoomFcn ~= 0)
                    eval(extZoomFcn);
                end
                return;
            otherwise
               % Zoom partially out
               m = -1;
               scale_factor = 2;
         end
      elseif strcmp(zoomMode,'out')
         switch selection_type
            case 'normal'
               % Zoom partially out
               m = -1;
               scale_factor = 2;
            case 'open'
               % Zoom all the way out
               zoom(fig,'out');
               return;
            otherwise
               % Zoom in
               m = 1;
               scale_factor = 2; % the default zooming
(cont.)  factor
         end
      else % unrecognized zoomMode
         return
      end


      ZOOM_Pt1 = get_currentpoint(ax);
      ZOOM_Pt2 = ZOOM_Pt1;
      center = ZOOM_Pt1;

      if (m == 1)
         % Zoom in
         units = get(fig,'units');
(cont.)  set(fig,'units','pixels')

         rbbox([get(fig,'currentpoint') 0
(cont.)  0],get(fig,'currentpoint'));

         ZOOM_Pt2 = get_currentpoint(ax);
         set(fig,'units',units)

         % Note the currentpoint is set by having a
(cont.)  non-trivial up function.
         if min(abs(ZOOM_Pt1-ZOOM_Pt2)) >= ...
               min(.01*[diff(get_xlim(ax))
(cont.)  diff(get_ylim(ax))]),
            % determine axis from rbbox
            a = [ZOOM_Pt1;ZOOM_Pt2]; a =
(cont.)  [min(a);max(a)];

            % Undo the effect of get_currentpoint for log
(cont.)  axes
            if strcmp(get(ax,'XScale'),'log'),
               a(1:2) = 10.^a(1:2);
            end
            if strcmp(get(ax,'YScale'),'log'),
               a(3:4) = 10.^a(3:4);
            end
            rbbox_mode = 1;
         end
      end
      limits = zoom(fig,'getlimits');

case 'scale',
      if all(get(ax,'view')==[0 90]), % 2D zooming with
(cont.)  scale_factor

         % Activate axis that is clicked in
         ZOOM_found = 0;
         ax = gca;
         xlim = get(ax,'xlim');
         ylim = get(ax,'ylim');
         ZOOM_Pt1 = [sum(xlim)/2 sum(ylim)/2];
         ZOOM_Pt2 = ZOOM_Pt1;
         center = ZOOM_Pt1;

         if (xlim(1) <= ZOOM_Pt1(1,1) & ZOOM_Pt1(1,1) <=
(cont.)  xlim(2) & ...
               ylim(1) <= ZOOM_Pt1(1,2) & ZOOM_Pt1(1,2)
(cont.)  <= ylim(2))
            ZOOM_found = 1;
         end % if

         if ZOOM_found==0, return, end

         if (scale_factor >= 1)
            m = 1;
         else
            m = -1;
         end

      else % 3D
         old_CameraViewAngle = get(ax,
(cont.) ,'CameraViewAngle')*pi/360;
         ncva = atan(tan(old_CameraViewAngle)
(cont.) )*(1/scale_factor))*360/pi;
```

```
      set(ax,'CameraViewAngle',ncva);
      return;
   end

   limits = zoom(fig,'getlimits');

case 'getmode'
   state = getappdata(fig,'ZOOMFigureState');
   if isempty(state)
      out = 'off';
   else
      mode = getappdata(fig,'ZOOMFigureMode');
      if isempty(mode)
         out = 'on';
      else
         out = mode;
      end
   end
   return


case 'on',
   state = getappdata(fig,'ZOOMFigureState');
   if isempty(state),
       % turn off all other interactive modes
       state = uiclearmode(fig,'zoom',fig,'off');

       % restore button down functions for any other
(cont.)  children of the figure
       uirestore(state,'children');
       setappdata(fig,'ZOOMFigureState',state);
   end

   set(findall(fig,'Tag','figToolZoomIn'),'State',
(cont.) ,'on');
   set(fig,'windowbuttondownfcn','zoom(gcbf,''down'')',
(cont.) , ...
           'windowbuttonupfcn','ones;', ...
           'windowbuttonmotionfcn','', ...
           'buttondownfcn','', ...
```

```
                ’interruptible’,’on’);
        set(ax,’interruptible’,’on’)
        return

    case ’inmode’
        zoom(fig,’on’);
        set(findall(fig,’Tag’,’figToolZoomIn’),’State’,
(cont.) ,’on’);
        set(findall(fig,’Tag’,’figToolZoomOut’),’State’,
(cont.) ,’off’);
        setappdata(fig,’ZOOMFigureMode’,’in’);
        return

    case ’outmode’
        zoom(fig,’on’);
        set(findall(fig,’Tag’,’figToolZoomIn’),’State’,
(cont.) ,’off’);
        set(findall(fig,’Tag’,’figToolZoomOut’),’State’,
(cont.) ,’on’);
        setappdata(fig,’ZOOMFigureMode’,’out’);
        return

    case ’reset’,
        axz = get(ax,’ZLabel’);
        if isappdata(axz,’ZOOMAxesData’)
          rmappdata(axz,’ZOOMAxesData’);
        end
        return

    case ’xon’,
        zoom(fig,’on’) % Set up userprop
        set(fig,’windowbuttondownfcn’,’zoom(gcbf,
(cont.) ,’’xdown’’)’, ...
            ’windowbuttonupfcn’,’ones;’, ...
            ’windowbuttonmotionfcn’,’’,’buttondownfcn’,’’,
(cont.) ,...
            ’interruptible’,’on’);
        set(ax,’interruptible’,’on’)
        return
```

```
case ’yon’,
   zoom(fig,’on’) % Set up userprop
   set(fig,’windowbuttondownfcn’,’zoom(gcbf,
(cont.) ,’’ydown’’)’, ...
      ’windowbuttonupfcn’,’ones;’, ...
      ’windowbuttonmotionfcn’,’’,’buttondownfcn’,’’,
(cont.) ,...
      ’interruptible’,’on’);
   set(ax,’interruptible’,’on’)
   return

case ’out’,
   limits = zoom(fig,’getlimits’);
   center = [sum(get_xlim(ax))/2
(cont.)  sum(get_ylim(ax))/2];
   m = -inf; % Zoom totally out

case ’getlimits’, % Get axis limits
   axz = get(ax,’ZLabel’);
   limits = getappdata(axz,’ZOOMAxesData’);
   % Do simple checking of userdata
   if size(limits,2)==4 & size(limits,1)<=2,
      if all(limits(1,[1 3])<limits(1,[2 4])),
         getlimits = 0; out = limits(1,:); return   %
(cont.)  Quick return
      else
         getlimits = -1; % Don’t munge data
      end
   else
      if isempty(limits), getlimits = 1; else
(cont.)  getlimits = -1; end
   end

   % If I’ve made it to here, we need to compute
(cont.)  appropriate axis
   % limits.

   if isempty(getappdata(axz,’ZOOMAxesData’)),
      % Use quick method if possible
      xlim = get_xlim(ax); xmin = xlim(1); xmax =
(cont.)  xlim(2);
      ylim = get_ylim(ax); ymin = ylim(1); ymax =
(cont.)  ylim(2);

   elseif strcmp(get(ax,'xLimMode'),'auto') & ...
         strcmp(get(ax,'yLimMode'),'auto'),
      % Use automatic limits if possible
      xlim = get_xlim(ax); xmin = xlim(1); xmax =
(cont.)  xlim(2);
      ylim = get_ylim(ax); ymin = ylim(1); ymax =
(cont.)  ylim(2);

   else
      % Use slow method only if someone else is using
(cont.)  the userdata
      h = get(ax,'Children');
      xmin = inf; xmax = -inf; ymin = inf; ymax =
(cont.)  -inf;
      for i=1:length(h),
         t = get(h(i),'Type');
         if ~strcmp(t,'text'),
            if strcmp(t,'image'), % Determine axis
(cont.)  limits for image
               x = get(h(i),'Xdata'); y =
(cont.)  get(h(i),'Ydata');
               x = [min(min(x)) max(max(x))];
               y = [min(min(y)) max(max(y))];
               [ma,na] = size(get(h(i),'Cdata'));
               if na>1, dx = diff(x)/(na-1); else dx =
(cont.)  1; end
               if ma>1, dy = diff(y)/(ma-1); else dy =
(cont.)  1; end
               x = x + [-dx dx]/2; y = y + [-dy
(cont.)  dy]/2;
            end
            xmin = min(xmin,min(min(x)));
            xmax = max(xmax,max(max(x)));
            ymin = min(ymin,min(min(y)));
            ymax = max(ymax,max(max(y)));
         end
      end

      % Use automatic limits if in use (override
(cont.)  previous calculation)
      if strcmp(get(ax,'xLimMode'),'auto'),
         xlim = get_xlim(ax); xmin = xlim(1); xmax =
(cont.)  xlim(2);
      end
      if strcmp(get(ax,'yLimMode'),'auto'),
         ylim = get_ylim(ax); ymin = ylim(1); ymax =
(cont.)  ylim(2);
      end
   end

   limits = [xmin xmax ymin ymax];
   if getlimits~=-1, % Don't munge existing data.
      % Store limits ZOOMAxesData
      % store it with the ZLabel, so that it's cleared
(cont.)  if the
      % user plots again into this axis.  If that
(cont.)  happens, this
      % state is cleared
      axz = get(ax,'ZLabel');
      setappdata(axz,'ZOOMAxesData',limits);
   end

   out = limits;
   return

case 'getconnect', % Get connected axes
   axz = get(ax,'ZLabel');
   limits = getappdata(axz,'ZOOMAxesData');
   if all(size(limits)==[2 4]), % Do simple checking
      out = limits(2,[1 2]);
   else
      out = [ax ax];
   end
   return

case 'fill',
   old_view = get(ax,'view');
   view(45,45);
   set(ax,'CameraViewAngleMode','auto');
   set(ax,'CameraViewAngle',get(ax,
(cont.) ,'CameraViewAngle'));
   view(old_view);
   return

otherwise
   error(['Unknown option: ',zoomCommand,'.']);
end

%
% Actual zoom operation
%

if ~rbbox_mode,
   xmin = limits(1); xmax = limits(2);
   ymin = limits(3); ymax = limits(4);

   if m==(-inf),
      dx = xmax-xmin;
      dy = ymax-ymin;
   else
      dx = diff(get_xlim(ax))*(scale_factor.^(-m-1));
(cont.)  dx = min(dx,xmax-xmin);
      dy = diff(get_ylim(ax))*(scale_factor.^(-m-1));
(cont.)  dy = min(dy,ymax-ymin);
   end

   % Limit zoom.
   center = max(center,[xmin ymin] + [dx dy]);
   center = min(center,[xmax ymax] - [dx dy]);
   a = [max(xmin,center(1)-dx) min(xmax,center(1)+dx)
(cont.)  ...
         max(ymin,center(2)-dy)
(cont.)  min(ymax,center(2)+dy)];

   % Check for log axes and return to linear values.
   if strcmp(get(ax,'XScale'),'log'),
      a(1:2) = 10.^a(1:2);
   end
   if strcmp(get(ax,'YScale'),'log'),
      a(3:4) = 10.^a(3:4);
   end

end

% Check for axis equal and update a as necessary
if strcmp(get(ax,'plotboxaspectratiomode'),'manual') &
(cont.)  ...
   strcmp(get(ax,'dataaspectratiomode'),'manual')
   ratio = get(ax,'plotboxaspectratio')./get(ax,
(cont.) ,'dataaspectratio');
   dx = a(2)-a(1);
   dy = a(4)-a(3);
   [kmax,k] = max([dx dy]./ratio(1:2));
   if k==1
      dy = kmax*ratio(2);
      a(3:4) = mean(a(3:4))+[-dy dy]/2;
   else
     dx = kmax*ratio(1);
     a(1:2) = mean(a(1:2))+[-dx dx]/2;
   end
end

% Update circular list of connected axes
list = zoom(fig,'getconnect'); % Circular list of
(cont.)  connected axes.
if zoomx,
   if a(1)==a(2), return, end % Short circuit if zoom
(cont.)  is moot.
   set(ax,'xlim',a(1:2))
   h = list(1);
   while h ~= ax,
      set(h,'xlim',a(1:2))
      % Get next axes in the list
      hz = get(h,'ZLabel');
      next = getappdata(hz,'ZOOMAxesData');
      if all(size(next)==[2 4]), h = next(2,1); else h
```

```
(cont.)  = ax; end
   end
end
if zoomy,
   if a(3)==a(4), return, end % Short circuit if zoom
(cont.)  is moot.
   set(ax,'ylim',a(3:4))
   h = list(2);
   while h ~= ax,
      set(h,'ylim',a(3:4))
      % Get next axes in the list
      hz = get(h,'ZLabel');
      next = getappdata(hz,'ZOOMAxesData');
      if all(size(next)==[2 4]), h = next(2,2); else h
(cont.)  = ax; end
   end
end

% Evaluate post-zoom callback, to update graph labels
(cont.)  as necessary. [DLowd]
if zoomx | zoomy,
    handles = guidata(fig);
    zoomFcn = handles.postZoomFcn;
    if (zoomFcn ~= 0),
        eval(zoomFcn);
    end
end

function p = get_currentpoint(ax)
%GET_CURRENTPOINT Return equivalent linear scale
(cont.)  current point
p = get(ax,'currentpoint'); p = p(1,1:2);
if strcmp(get(ax,'XScale'),'log'),
   p(1) = log10(p(1));
end
if strcmp(get(ax,'YScale'),'log'),
   p(2) = log10(p(2));
end

function xlim = get_xlim(ax)
```

```
%GET_XLIM Return equivalent linear scale xlim
xlim = get(ax,'xlim');
if strcmp(get(ax,'XScale'),'log'),
   xlim = log10(xlim);
end

function ylim = get_ylim(ax)
%GET_YLIM Return equivalent linear scale ylim
ylim = get(ax,'ylim');
if strcmp(get(ax,'YScale'),'log'),
   ylim = log10(ylim);
end
```

Bibliography

Duda, R. and Hart, P. (1973). *Pattern Classification and Scene Analysis*. John Wiley and Sons, Menlo Park, California.

Epter, S., Krishnamoorthy, M., and Zaki, M. (2000). Clusterability detection and cluster initialization. pages 47–58.

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (2000). A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, 2.

Halkidi, M., Batistakis, Y., and Vazirgiannis, M. (2001a). Clustering algorithms and validity measures. *Proceedings of SSDBM Conference*.

Halkidi, M., Batistakis, Y., and Vazirgiannis, M. (2001b). Clustering validity checking methods: Part ii. *JIIS*, 17:107–145.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999a). Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999b). Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323.

Theodoridis, S. and Koutroumbas, K. (2003). *Pattern Recognition*. Academic Press, New York, NY.