Sandia National Laboratories: Implementation of and Experimenting with a Clustering Tool

Team: Avani Gadani, Daniel Lowd, Brian Roney, Eric Wu
Harvey Mudd College

Dr. Belinda Thom (Advisor)
Dr. Kevin Boyack (Liaison)

May 8, 2003
Contents

1 Introduction 1

2 Background 2
  2.1 Problem Definition ........................................ 2

3 Approach 3
  3.1 Goals .................................................. 4

4 Management 6
  4.1 Timeline .................................................. 6
  4.2 Team Member Contributions .............................. 6

5 Tool Development Approach 7
  5.1 Development Environment .................................. 7
  5.2 Interface Description ...................................... 7
  5.3 Code Structure ........................................... 8

6 Future Plans 9
  6.1 Proposed GUI changes ..................................... 9
  6.2 Validity .................................................. 9

A Clustering Algorithms 11
  A.1 k-means .................................................. 11
  A.2 Single-Linkage ........................................... 11

B Screenshots 13

C Source Code 17
  C.1 controller.m ............................................. 17
  C.2 plotwindow.m ............................................ 24
  C.3 clustercontrol.m ......................................... 33
  C.4 selectcontrol.m .......................................... 37
Chapter 1

Introduction

Clustering is a tool for better understanding a given set of data. It has uses in various informatics fields such as genomics and cross-citation analysis that tend to generate data sets large enough that it’s difficult for humans to visually correlate. We have implemented a tool that allows side by side comparison of different dimension reductions of a given data set, as well as the ability to cluster points using a k-means clustering and a single-link clustering algorithm. The code is generic, so more clustering algorithms may be added in the future. The tool is written in MATLAB for its rapid prototyping ability and was designed to function on data sets of 500 to 10,000 two-dimensional data points as requested by our sponsor.

The two currently implemented clustering algorithms were chosen for their simplicity, speed, and ability to handle varying types of data sets. K-means clustering is a partitional algorithm which runs quickly but tends to result in clusters with non-convex shapes [3]. Single-link clustering, on the other hand, is a hierarchical algorithm that can deal with arbitrary shapes, potentially at the expense of simple clusters[3].
Chapter 2

Background

Sandia works in various fields that require careful analysis and correlation of data, e.g. they have data points in infant leukemia research, and they’d like to determine similar genes to use medicines on. Correlation of titles and abstracts to institutions in research papers published from various sources would show trends in development. Researchers at Sandia desire a tool that would allow them to visually compare multiple ordinations, analyze them, and that would calculate various validity metrics for each ordination.

We’re approaching this problem through clustering. In clustering, the object is to place elements into the same cluster when they are similar enough according to some predefined metric. The predefined metric is one aspect that makes clustering a subjective process. There are two types of clustering: 1) partitional clustering, where every point is assigned to exactly one cluster, and 2) hierarchical clustering, where every cluster of size greater than one circumscribes a smaller group inside of it [2].

2.1 Problem Definition

Sandia needs to be able to view scatter plots of different dimensionally reduced data sets. Additionally, Sandia would like to be able to apply a clustering on different scatter plots in order to aid in comparing how similar various ordinations are. Also, user intervention on automatic clustering is desired in order to see how the clustering applies to scatterplots generated using other algorithms. Finally, we wish to provide a variety of clustering metrics in order to aid users in assessing a particular datasets clusterability.

Ease of use and generality is considered important in the design. We have chosen to provide a GUI interface, as well as for many of the key functions to be accessible through the MATLAB command line.
Chapter 3

Approach

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 automatic clustering, nor to human insight. The dimensionality of the data (i.e., number of features) is likely to be too high to be easily visualized by a person. However, the data also cannot be automatically explored yet, due to problems in determining how to compare various features. 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. Our tool can run various clustering algorithms (currently k-means and single-link) to cluster the ordination from VxOrd. We 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 us 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
Figure 3.1: 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.

remain together in ordinations based on keywords in titles and co-citations, for example, then we know that articles with similar keywords in titles are likely to cite the same papers. In addition, the tool should support various validity metrics in the future. The side-by-side comparison of clusterings in addition to the validity metrics should suggest ordinations that should be studied in further detail in VxInsight.

As shown in Figure 3.1, 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 3.2 human investigation of clusters in detail can begin.

3.1 Goals

The main goals of the tool are

- To allow a user to visually compare different coordinate files produced using different features or random seeds.
- To provide functionality for clustering the data using several algorithms, and to automatically compare clusterings using known validation methods.
- To ultimately allow the user and the clustering algorithm to work together to produce a better clustering solution than either could individually.
Figure 3.2: Extension to process, for comparing different similarity metrics. The tool we propose to
develop will read multiple ordinations generated from different similarity files, which in turn were
generated using different similarity measures.

- To allow a user to experiment with different validity metrics over clusterings.

Our tool is currently MATLAB-based and for Microsoft Windows that will have file-level
interaction with vxOrd and possibly vxInsight.

The actual layout of the tool will be described later, but essentially the tool displays scatterplots
of data sets under different clustering algorithms or different similarity metrics from vxOrd. A
user may manually recluster portions of a data set when automatic clustering algorithms perform
poorly. Also, a user may apply metrics to the clustered plots to automatically determine the relative
quality of a given clustering.

To keep the transition from vxInsight to our tool simple, we need to be able to identify and
refer back to clusterings of points of the plot from vxInsight. One way to do this would be the use
of a clustering algorithm such as k-means.

Currently, both k-means and single-linkage clustering algorithms (See Appendix A) are imple-
mented in our tool. We hope to add clustering algorithms and optimize them in C before delivering
our product.
Chapter 4

Management

4.1 Timeline

Our current plan is to meet the following deadlines:

- 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. 25th: Poster Due
- May 9th: Final Report/CD

4.2 Team Member Contributions

We’ve currently divided the project so that Daniel and Brian are working on the coding end of the problem while Eric and Avani are exploring the research aspect. As soon as the tool reaches a stable feature set, the entire team intends to become involved in the research aspect of this project.

In particular, Daniel is going to focus on tool optimizations and integrations, Brian is going to explore data sets, and Eric and Avani are going to look into cluster validity (Chapter 6)
Chapter 5

Tool Development Approach

5.1 Development Environment

To quickly develop an extensible solution, we used MATLAB as our programming language and development platform. The main disadvantages of MATLAB are that it makes some restrictions on GUI development and may be somewhat slower than an implementation in C or C++. In particular, single-link is very slow, so we hope to optimize the built-in MATLAB algorithm using C. However, there’s a large amount of functionality built-in to MATLAB, including several clustering algorithms and plotting tools.

5.2 Interface Description

In our proposal we had the following Project Design Specifications:

We shall implement a function in MATLAB that will generate a GUI. This GUI will include:

- A file browsing system to select data sets to examine, each of which will be placed in a separate plot window.
- The ability to zoom, scale, and/or rotate any of the plots individually.
- The ability to select a subset of a plot, which will highlight the equivalent points in the other plot windows.
- User can select an unlimited number of .coord files for comparison.

All of these features, except for plot rotation, have already been implemented. In addition, users may cluster any ordination with either the k-means or the single linkage clustering algorithm. The user also specifies the number of clusters to produce. Each cluster in the ordination is plotted in a different color and symbol combination. MATLAB will plot points in any of 7 colors and 13 symbols, yielding 91 potential combinations. When one ordination is clustered, all ordination plots are recolored to reflect the clustering. Users may also perform a manual clustering by specifying a color, a symbol, and a set of points in one of the plots.
5.3 Code Structure

Our tool consists of four classes, each one responsible for the operations of one window type in the tool. The Controller manages the other windows, including windows for ordination plots and dialogs for clustering and point selection. The PlotWindow class colors and displays a single ordination. It communicates with the Controller through a number of callbacks. The ClusterControl class allows the user to specify the ordination to cluster and the algorithm and parameters to use for the clustering. This information is passed onto the Controller in a callback, which calls the actual clustering algorithm and tells each PlotWindow to re-color itself according to the new clustering. The SelectControl class lets the user specify the color and symbol type to color selected regions. None of PlotWindow, ClusterControl, and SelectControl ever references any class but Controller. This structure, with the Controller responsible for managing the other objects, simplifies design and implementation and results in code that is easier to maintain. This will be helpful for implementing additional requirements, such as the validity metrics.
Chapter 6

Future Plans

6.1 Proposed GUI changes

– A function that allows you to pick a point in a clustered plot and zoom onto the cluster that point is in.
– A Command Line Interface so that we can script our experiments.
– The ability to save and load results.
– Incorporate Validity.
– Cleaner and more optimized code.

6.2 Validity

Another goal for next semester is to research and implement a variety of validity metrics. It may be difficult to determine whether an ordination has any clusterings, or to decide which of several clusterings is more meaningful.

To determine whether an ordination can be clustered usefully, we intend to implement geometric comparisons such as nearest neighbor and Hubert’s V statistic on the ordination. While this cannot solve the problem entirely, it will help to differentiate ordinations from each other and from randomly generated data [5].

Geometric validity measures focus on whether a representation of a data set is sufficiently non-random to have meaningful clusters at all, while cluster-based metrics look at the clusters generated [2]. Using Monte Carlo methods to create random data sets, we can use geometric measures to statistically compare ordinations to randomly distributed data. If the difference between the values of the validity measure for the random data and the ordination is too small, then the ordination may be too random to cluster effectively [4]. Cluster-based metrics such as the Bailey-Dubes index attempt to determine how well the data fits the pattern expected by a given clustering algorithm, indicating whether or not that algorithm provides a good clustering for a given data set.
Figure 6.1: A possible clustering

There is no conclusive answer to the clustering problem, as everyone has a different idea of what a “good” clustering is. Validity metrics are one method of attempting to objectively determine the quality of a clustering [5]. Ideally, given many different ordinations of a given data set, using many different similarity computations, validity metrics would help automatically suggest similarities and ordinations that are worthy of further study using vxInsight.

Fig 6.1 demonstrates why the clustering problem is hard, as 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, a validity metric may assist in determining which clustering fits the data best. However, cluster validity is 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.
Appendix A

Clustering Algorithms

A.1 k-means

k-means clustering is a commonly used partitional algorithm. 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

\[ E = \sum_{i=1}^{c} \sum_{x \in C_i} d(x, m_i) \]  \hspace{1cm} (A.1)

, where \( m_i \) is the center of cluster \( C_i \).

The run-time for the k-means algorithm is \( O(k \cdot n) \) 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 [1]. 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/Murty/Flynn, page 18). An inherent issue in k-means clustering is that convex clusters tend to be produced when Euclidean distance is used as the distance measure.

A.2 Single-Linkage

Single-link clustering can broaden one’s search to include non-convex clusters, though the algorithm introduces other issues. 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 $O(n^2 \log n)$. 

Figure A.1: Example of a scatterplot with non-convex clusters
Figure B.1: The Controller GUI. The Open button is used for loading ordinations. The Cluster and Select buttons open separate dialogs for automatically or manually clustering ordinations, respectively. The Zoom button toggles zoom mode on all plots. When zoom mode is on, left-clicking on a plot zooms in and right-clicking zooms out.

Figure B.2: The Cluster dialog. Once the user has specified the ordination to cluster (from the pop-up menu), the algorithm to use, and any additional algorithm parameters, clicking on the Cluster button runs the algorithm. Though the clustering is done according to the selected ordination, all plots are recolored accordingly.
APPENDIX B. SCREENSHOTS

Figure B.3: A single plot window. Specifically, this shows an ordination from the ARIST data set clustered using k-means and 30 clusters. Different colors and symbols are used to distinguish one cluster from another.

Figure B.4: Another plot window. This shows a different ordination form the ARIST data set clustered using the single linkage algorithm with 30 clusters.
Figure B.5: The Select dialog. Permits the user to choose the symbol and color for manually changing selected points. When this dialog is open, clicking and dragging a box on an ordination plot will redraw all contained points using the color and symbol specified here. Corresponding points in other plots will be redrawn with this new color and symbol as well.
Appendix C

Source Code

C.1 controller.m

function varargout = controller(varargin)
% CONTROLLER Application M-file for controller.fig
% FIG = CONTROLLER launch controller GUI.
% CONTROLLER('callback_name', ...) invoke the named callback.
% Last Modified by GUIDE v2.0 10-Nov-2002 13:11:02

if nargin == 0 % LAUNCH GUI

    fig = openfig(mfilename,'reuse');

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

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

    handles.controlMode = 0;
    % what the controller thinks clicks on plots should do
    % 0 = nothing (default)
    % 1 = user selection of points
    % 2 = zoom
    % 3 = cluster

    handles.baseColor = 1;
    handles.highlightColor = 2;
% array of windows
handles.plots = [];
handles.plotnames = [];
handles.plothandles = [];

% temporary manipulation variables
handles.points = [];
handles.names = [];
handles.colors = [];
handles.filename = 0;

% Current path, for opening new files
handles.pathname = 0;

% FIX THIS - non implemented function
% changeSelectMode(fig, handles, 0);

% Update handles structure
guidata(fig, handles);

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

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

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

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

---

function varargout = openbutton_Callback(h, eventdata, handles, varargin)
% Stub for Callback of the uicontrol handles.openbutton1.
[handles.filename, handles.names, handles.points] = openButtonInternal;

len = length(handles.plots);
done = 0;
id = 1; % initial id value (increment until valid)
while ~done
    done =1;
    for i=1:len,
        if(handles.plots(i)== id) % if we find the id already in the array
            id = id+1; % increment the id
            done=0; % and try again
        end;
    end;
end;
% we now have a unique id
handles.plots = [handles.plots, id]; % so put it in the array
handles.plotnames = strvcat(handles.plotnames, handles.filename);
    % keep track of the name of this plot

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

if(handles.filename ~=0)
    f = plotwindow;
    plotwindow('init', f, h, handles.filename, handles.names, handles.points, ...
             handles.colors, id);

    handles.plothandles= [handles.plothandles, f];
end;
guida(h, handles);

% -------------------------------

function [filename, names, points] = openButtonInternal

% Stub for Callback of the uicontrol handles.openbutton1.

names = 0;
points = 0;

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

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

% Save the directory
cd(pathname);

% Get points
[names, xcoords, ycoords] = textread(filename, '%s%f%f');
points = [xcoords, ycoords];
function varargout = clusterbutton_Callback(h, eventdata, handles, varargin)
    handles = guidata(h);
    clustercontrol('init', handles.plotnames, h);

    % function varargout = activecluster(h, name, iterations, numclusters, clusteralg)
    handles = guidata(h);
    len = length(handles.plots);
    for i = 1:len,
        if(strcmp(handles.plotnames(i, :), name))
            if(clusteralg==0)
                [filename, names, points, colors, id] = plotwindow('getInfo', handles.plothandles(i));
                clusters = kmeans(points, numclusters, 'Maxiter', iterations, ...
                    'EmptyAction', 'singleton');
                for j = 1:len,
                    plotwindow('recolor', handles.plothandles(j), names, clusters);
                end;
            end;
            break;
        end;
    end;
    break;
end;

function varargout = togglebutton3_Callback(h, eventdata, handles, varargin)
    if (get(handles.togglebutton4, 'Value') == 1 & & ... 
        get(handles.togglebutton3, 'Value') == 1)
        set(handles.togglebutton4, 'Value', 0);
        togglebutton4_Callback(h, 0, handles);
    end
if (get(handles.togglebutton3, 'Value') == 1)
    % Open selection GUI
    handles.selectfig = selectcontrol('init', h);
    handles.controlMode = 1;
else
    close(handles.selectfig);
    handles.controlMode = 0;
end

guidata(h, handles);

% -------------------------------------------------------------------
-
function selectClosingCallback(h)

% Switch out of selection mode when required GUI is closed
handles = guidata(h);
set(handles.togglebutton3, 'Value', 0);
handles.controlMode = 0;
guidata(h, handles);

% -------------------------------------------------------------------
-
function setSelectColor(h, c)
handles = guidata(h);
handles.selectColor = c;
guidata(h, handles);

% -------------------------------------------------------------------
-
function mode = getMode(h)
handles = guidata(h);
mode = handles.controlMode;

% -------------------------------------------------------------------
-
function plotPointSelectionCallback(h, names)
handles = guidata(h);
len = length(handles.plots);
for i = 1:len,
    plotwindow('select', handles.plothandles(i), names, handles.selectColor);
end;
% ---------------------------------------------------------------
function varargout = togglebutton4_Callback(hObject, eventdata, handles, varargin)

if (get(handles.togglebutton4, 'Value') == 1 && ...
    get(handles.togglebutton3, 'Value') == 1)
    set(handles.togglebutton3, 'Value', 0);
    togglebutton3_Callback(hObject, 0, handles);
end

len = length(handles.plots);
h = gcf;
for i = 1:len,
    if (get(handles.togglebutton4, 'Value') == 1)
        zoom(handles.plots(i), 'ON');
    else
        zoom(handles.plots(i), 'OFF');
    end;
end

figure(gcf);

% ---------------------------------------------------------------
function varargout = plotClosingCallback(hObject, id)
handles = guidata(hObject); % get handle data back again
len = length(handles.plots); % how many plots we had open before
n=1;

for i = 1:len,
    if(handles.plots(i) == id)
        n=i; % find the index of the array that matches filename
        end;
end;

handles.plots = [handles.plots(1:n-1), handles.plots(n+1:len)]; % and re-
move it from the array
handles.plotnames = strvcat(handles.plotnames(1:n-1,:), handles.plotnames(n+1:len,:);
handles.plothandles = [handles.plothandles(1:n-1), handles.plothandles(n+1:len)];

guidata(hObject, handles); % save changes to handles
C.2  plotwindow.m

function varargout = plotwindow(varargin)
% PLOTWINDOW Application M-file for plotwindow.fig
% FIG = PLOTWINDOW launch plotwindow GUI.
% PLOTWINDOW(‘callback_name’, ...) invoke the named 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,’defaultUicontrolBackgroundColor’));

    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    handles.parent = 0;
    handles.filename = 0;
    handles.names = 0;
    handles.points = 0;
    handles.colors = 0;
    handles.id = 0;

    guidata(fig, handles);

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

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

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

% -------------------------------------------------------------------
function init(h, parent, filename, names, points, colors, id)
% Set the title bar to the filename
set(h, 'Name', filename);

% Set all the handles data
handles = guidata(h);
handles.parent = parent;
handles.filename = filename;
handles.names = names;
handles.points = points;
handles.colors = colors;
handles.id = id;
guidata(h, handles);

% Plot the graph
redraw(h);

% -------------------------------------------------------------------
-

function [filename, names, points, colors, id] = getInfo(h)
handles = guidata(h);
filename = handles.filename;
names = handles.names;
points = handles.points;
colors = handles.colors;
id = handles.id;

% -------------------------------------------------------------------
-

function recolor(h, names, colors)
handles = guidata(h);
baseColors = zeros(length(handles.names),1)-1;
handles.colors = fastcolor(handles.names, names, colors, baseColors);
guidata(h, handles);
redraw(h);

% -------------------------------------------------------------------
-

function colors2 = mergecolor(names2, names1, colors1, baseColors)

colors2 = 0;

if (length(names2) == 0 || length(names1) == 0)
colors2 = baseColors;
return;
end

for idx1 = ceiling(length(names1)/2):length(names1)
    for idx2 = 1:length(names2)
        if strcmp(names1(idx1), names2(idx2))
            before = mergecolor(names2(1:idx2-1), names1(1:idx1-1), ...
                    colors1(1:idx1-1), baseColors(1:idx2-1));
            after = mergecolor(names2(idx2+1:length(names2)), ...
                    names1(idx1+1:length(names1)), ...
                    colors1(idx1+1:length(names1)), baseColors(idx2+1:length(names2)));
            colors2 = [before; colors1(idx1); after];
            return;
        end
    end
end

colors2 = mergecolor(names2, names1(1:(ceiling(length(names1)/2) - 1)), colors1, colors2);

% ---------------------------------------------------------------
% function colors2 = mergecolor2(names2, names1, colors1, baseColors)
%     colors2 = mergecolor2rec(names2, names1, colors1, baseColors, ...
%     1, length(names1), 1, length(names2));
% ---------------------------------------------------------------

function colors2 = mergecolor2rec(names2, names1, colors1, baseColors, ...
    min1, max1, min2, max2)

colors2 = baseColors;
if (max1 - min1 < 0 || max2 - min2 < 0)
    return;
end

% Start halfway between min1 and max1
idx1 = ceiling((min1 + max1)/2);

while (idx1 <= max1)
    for idx2 = min2:max2
        if strcmp(names1(idx1), names2(idx2))
            before = mergecolor(names2(1:idx2-1), names1(1:idx1-1), ...
                    colors1(1:idx1-1), baseColors(1:idx2-1));
            after = mergecolor(names2(idx2+1:length(names2)), ...
                    names1(idx1+1:length(names1)), ...
                    colors1(idx1+1:length(names1)), baseColors(idx2+1:length(names2)));
            colors2 = [before; colors1(idx1); after];
            return;
        end
    end
end

APPENDIX C. SOURCE CODE

colors2(idx2) = colors1(idx1);
colors2 = mergecolor2rec(names2, names1, colors1, colors2, ...
    min1, idx1-1, min2, idx2-1);
colors2 = mergecolor2rec(names2, names1, colors1, colors2, ...
    idx1+1, max1, idx2+1, max2);
return;
end
end

idx1 = idx1 + 1;
end

colors2 = mergecolor2rec(names2, names1, colors1, colors2, ...
    min1, min1 + ceil((max1-min1)/2) - 1, min2, max2);

% --------------------------------------------------------------------------------------------------
- function colors2 = slowcolor(names2, names1, colors1, baseColors)

colors2 = baseColors;
idx2 = 1;
for idx1 = 1:length(names1)
    for off2 = idx2:length(names2)
        if (strcmp(names1(idx1), names2(off2)))
            colors2(off2) = colors1(idx1);
            idx2 = off2 + 1;
        end
    end
    idx1 = idx1 + 1;
end

% --------------------------------------------------------------------------------------------------
- function colors2 = fastcolor(names2, names1, colors1, baseColors)

MAX_OFFSET = 500;
colors2 = baseColors;
idx1 = 1;
idx2 = 1;

while (idx1 <= length(colors1) & idx2 <= length(colors2))
    if (idx1 == length(colors1))
        a = 1;
    else
        a = 2;
    end
    ...
APPENDIX C. SOURCE CODE

```matlab
end
for k = 0:MAX_OFFSET
    % Ensure that our experimental offsets don’t 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 + k2))
            break;
        end
    end

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

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

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

    if (k == MAX_OFFSET)
        disp('Encountered section of MAX_OFFSET ID’s with no match. Aborting');
        break;
    end
```

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

% Update the plot in this window
function redraw(h)
handles = guidata(h);
axes(handles.axes1);
%scaling = axis;
cla;

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

xLists = cell(numColors, numSymbols);
yLists = cell(numColors, numSymbols);

% Treat black points specially
xBlack = cell(1, numSymbols);
yBlack = cell(1, numSymbols);

colors = handles.colors;
numpoints = size(handles.points);
for i = 1:numpoints
    % A color of -sym => black
    if (colors(i) < 0)
        sym = -colors(i);
        xBlack{1,sym} = [xBlack{1,sym}, handles.points(i,1)];
        yBlack{1,sym} = [yBlack{1,sym}, handles.points(i,2)];
    else
        col = mod(colors(i)-1,numColors) + 1;
        sym = mod(floor((colors(i)-1)/numColors), numSymbols) + 1;
        xLists{col,sym} = [xLists{col,sym}, handles.points(i, 1)];
        yLists{col,sym} = [yLists{col,sym}, handles.points(i, 2)];
    end
end

for sym = 1:numSymbols
    for col = 1:numColors
        hold on
plot(xLists{col,sym}, yLists{col,sym}, [colorStrs(col), symbolStrs(sym)]);
end

% Plot the black points separately
hold on
plot(xBlack{1,sym}, yBlack{1,sym}, ["k", symbolStrs(sym)]);
end

%axis(scaling);

% ABOUT CALLBACKS:
% GUIDE automatically appends subfunction prototypes to this file, and
% sets objects’ callback properties to call them through the FEVAL
% switchyard above. This comment describes that mechanism.
% |
% Each callback subfunction declaration has the following form:
% |
% <SUBFUNCTION_NAME>(H, EVENTDATA, HANDLES, VARARGIN)
% |
% The subfunction name is composed using the object’s Tag and the
% callback type separated by ‘_’, e.g. ’slider2_Callback’,
% ’figure1_CloseRequestFcn’, ’axis1_ButtondownFcn’.
% |
% H is the callback object’s handle (obtained using GCBO).
% |
% EVENTDATA is empty, but reserved for future use.
% |
% HANDLES is a structure containing handles of components in GUI using
% tags as fieldnames, e.g. handles.figure1, handles.slider2. This
% structure is created at GUI startup using GUIHANDLES and stored in
% the figure’s application data using GUIDATA. A copy of the structure
% is passed to each callback. You can store additional information in
% this structure at GUI startup, and you can change the structure
% during callbacks. Call guidata(h, handles) after changing your
% copy to replace the stored original so that subsequent callbacks see
% the updates. Type ”help guihandles” and ”help guidata” for more
% information.
% |
% VARARGIN contains any extra arguments you have passed to the
% callback. Specify the extra arguments by editing the callback
% property in the inspector. By default, GUIDE sets the property to:
% |
% <MFILENAME>(’<SUBFUNCTION_NAME>’, gcbo, [], guidata(gcbo))
% |
% Add any extra arguments after the last argument, before the final
% closing parenthesis.
function varargout = axes1_ButtonDownFcn(h, eventdata, handles, varargin)
axes(handles.axes1);
if (controller('getMode', handles.parent) ~= 1)
    return;
end

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

selectedPoints = [];
% Get the selected region
numPoints = size(handles.points);
for i = 1:numPoints
    if (minPoint < handles.points(i,:)) ...
        & handles.points(i,:) < maxPoint)
        selectedPoints = [selectedPoints, handles.names(i)];
    end
end
controller('plotPointSelectionCallback', handles.parent, selectedPoints);

function select(h, names, color)
handles = guidata(h);
handles.colors = mergecolor(handles.names, names, ... 
    zeros(length(names)) + color, handles.colors);
guida(h, handles);
redraw(h);
function varargout = figure1_DeleteFcn(h, eventdata, handles, varargin)

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

% -------------------------------------------------------------------

function varargout = figure1_ReshapeFcn(h, eventdata, handles, varargin)

% TODO -- eventually we'll do the layout manually, to make it look nicer.
C.3 clustercontrol.m

function varargout = clustercontrol(varargin)
% CLUSTERCONTROL Application M-file for clustercontrol.fig
% FIG = CLUSTERCONTROL launch clustercontrol GUI.
% CLUSTERCONTROL(‘callback_name’, ...) invoke the named callback.

% Last Modified by GUIDE v2.0 17-Nov-2002 14:05:25
if nargin == 0 % LAUNCH GUI
    % See INIT - this should never be reached
elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK
    try
        if (nargout)
            [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
        catch
            disp(lasterr);
        end
    end

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

VARARGIN contains any extra arguments you have passed to the callback. Specify the extra arguments by editing the callback property in the inspector. By default, GUIDE sets the property to: <MFILENAME>(‘<SUBFUNCTION_NAME>’, gcbo, [], guidata(gcbo)) Add any extra arguments after the last argument, before the final closing parenthesis.

function init(plotnames, h)

% Set the title bar to the filename
    fig = openfig(mfilename,’reuse’);

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

    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    handles.numclusters = 30;
    handles.numiterations = 100;
    handles.clusteralg = 0;
    handles.plotnames = plotnames;
    handles.parent = h;
    % 0 = k-means
    % 1 = single-link
    set(handles.popupmenu1, ’String’, plotnames);

    guidata(fig, handles);

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

% -------------------------------------------------------------------
function varargout = popupmenu1_Callback(h, eventdata, handles, varargin)
    temp= get(handles.popupmenu1, 'String');
    value= get(handles.popupmenu1, 'Value');
    handles.nameplot= temp(value, :);

% -------------------------------------------------------------------

function varargout = singlelink_Callback(h, eventdata, handles, varargin)
    handles = guidata(h);
    set(handles.singlelink, 'Value', 1);
    set(handles.kmeans, 'Value', 0);
    handles.clusteralg = 1;
    guidata(h, handles);

% -------------------------------------------------------------------

function varargout = kmeans_Callback(h, eventdata, handles, varargin)
    handles = guidata(h);
    set(handles.kmeans, 'Value', 1);
    set(handles.singlelink, 'Value', 0);
    handles.clusteralg = 0;
    guidata(h, handles);

% -------------------------------------------------------------------

function varargout = clust_Callback(h, eventdata, handles, varargin)
    handles = guidata(h);
    handles.numclusters = str2num(get(handles.clust, 'String'));
    guidata(h, handles);

% -------------------------------------------------------------------

function varargout = iter_Callback(h, eventdata, handles, varargin)
    handles = guidata(h);
    handles.numiterations = str2num(get(handles.iter, 'String'));
    guidata(h, handles);

% -------------------------------------------------------------------

function varargout = clustergo_Callback(h, eventdata, handles, varargin)
    handles = guidata(h);
    temp = get(handles.popupmenu1, 'String');
    value = get(handles.popupmenu1, 'Value');
handles.nameplot = temp(value,:);
controller('activecluster', handles.parent, handles.nameplot, ...
    handles.numiterations, handles.numclusters, handles.clusteralg);
guidata(h, handles);
C.4  selectcontrol.m

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

% Last Modified by GUIDE v2.5 20-Nov-2002 19:05:54

if nargin == 0 % LAUNCH GUI
    fig = selectcontrol('init', 0);
    if nargout > 0
        varargout{1} = fig;
    end
elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK
    try
        if (nargout)
            [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
        catch
            disp(lasterr);
        end
    end
end

function fig = init(parent)
    fig = openfig(mfilename,"reuse");
    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));
    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    handles.parent = parent;
    handles.color = 1;
    handles.symbol = 1;
    guidata(fig, handles);
    notifyParent(fig, handles);
if nargout > 0
    varargout{1} = fig;
end

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

% function varargout = colorPopup_Callback(h, eventdata, handles, varargin)
handles.color = get(handles.colorPopup, 'Value');

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

% -------------------------------------------------------------------
-
function varargout = symbolPopup_Callback(h, eventdata, handles, varargin)

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

% -------------------------------------------------------------------
-
function varargout = notifyParent(h, handles)

if (handles.color == 7)
  controller('setSelectColor', handles.parent, -handles.symbol);
else
  controller('setSelectColor', handles.parent, ...
    (handles.symbol-1) * 6 + (handles.color));
end

% --- Executes during object deletion, before destroying properties.
function varargout = figure1_DeleteFcn(h, eventdata, handles, varargin)

controller('selectClosingCallback', handles.parent);
Bibliography

- A.K. Jain et. al. Data Clustering: A Review. ACM.
- Epter, Scott et. al. Clusterability Detection and Cluster Initialization. IBM Power Parallel Group: RPI