The Plot-Data Interface in Statistical Graphics

Catherine HURLEY*

Statistical software systems include modules for manipulating data sets, model fitting, and graphics. Because plots display data, and models are fit to data, both the model-fitting and graphics modules depend on the data. Today's statistical environments allow the analyst to choose or even build a suitable data structure for storing the data and to implement new kinds of plots. The multiplicity problem caused by many plot varieties and many data representations is avoided by constructing a plot-data interface. The interface is a convention by which plots communicate with data sets, allowing plots to be independent of the actual data representation. This article describes the components of such a plot-data interface. The same strategy may be used to deal with the dependence of model-fitting procedures on data.

Key Words: Abstraction barrier; Interactive plots; Software design.

1. INTRODUCTION

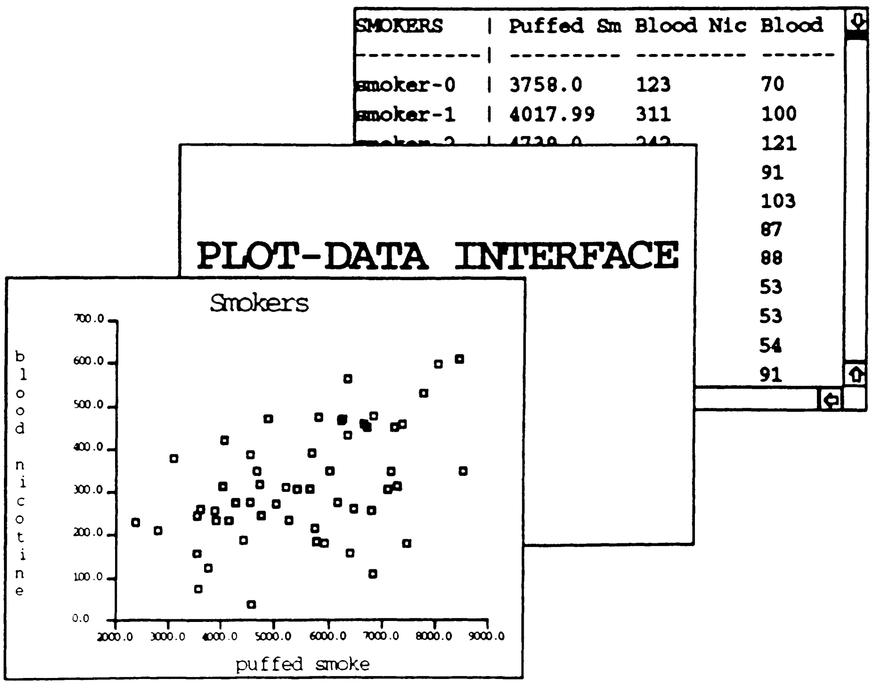
Today's advanced statistical software systems provide programming environments for data analysis. A statistical programming environment provides basic data structures and some built-in procedures for data manipulation, model fitting, and graphics. In addition, it provides a programming language with editing and debugging tools that allow the analyst to easily build new data representations specialized for the data set at hand, invent new graphical methods, and implement new model-fitting techniques, leading to still more data structures representing the fit results.

There are dependencies between the modules of a statistical system. Plots display data, and models are fit to data, so both the model-fitting and graphics modules depend on the data. It is important for implementors of new model-fitting modules, graphics functions, or data structures to understand the nature of this dependency so that new model-fitting techniques and graphical methods apply to existing data structures, and, conversely, existing fitting techniques and graphical methods apply to new data structures. In this article we study the dependency between plot and data.

We propose to handle the multiplicity problem caused by many plot varieties and many data representations by building an interface between plot and data and then iso-

*Assistant Professor, Department of Statistics, Computer and Information Systems, The George Washington University, Washington, DC 20052

©*1993 American Statistical Association, Institute of Mathematical Statistics, and Interface Foundation of North America*
Journal of Computational and Graphical Statistics, Volume 2, Number 4, Pages 365–379



```
SMOKERS    | Puffed Sm Blood Nic Blood
-----------| --------- --------- ------
smoker-0   | 3758.0    123       70
smoker-1   | 4017.99   311       100
                                 121
                                 91
                                 103
                                 87
                                 88
                                 53
                                 53
                                 54
                                 91
```

Figure 1. The Plot–Data Interface.

lating the dependencies into that interface. The interface will make it easier to construct plots of new data representations and to implement new data-display methods. We use two important software design concepts to construct the interface.

1. The *data abstraction* principle says that the use of a data object should be independent of its implementation (see, e.g., Abelson and Sussman 1985). In our situation the plot deals with an abstraction of the data set rather than the actual data structure used to represent the data set. The abstract version of the data is implemented by the plot-data interface, which contains the functions the plot uses to extract information from the data. As portrayed by Figure 1, the plot-data interface provides a barrier between a plot and the data set it represents, hiding the actual data set representation from the plot system.
2. *Generic functions* allow the same function to operate on different data types (see, e.g., Keene 1988). A generic function is actually a collection of *methods*. A method is like a function that operates only on objects of a particular type. When a generic function is invoked on an object, an automatic dispatch mechanism selects the appropriate method. By making the functions in the plot-data interface generic, the plot system can simultaneously deal with many different data set representations.

Section 2 describes conventional approaches to dealing with the plot-data dependency. Most simply, a plot function accepts data in a single, prescribed format—a scatterplot function that requires two data vectors, for example. The analyst must then preconvert from any other representation prior to plotting. This approach is unsatisfactory, overburdening the analyst and discouraging experimentation with new data structures.

Section 3 presents the very simple data model assumed by our plot-data interface. We emphasize that this model does not require a particular data representation. The interface is developed in the context of the Views model for statistical graphics (Hurley and Oldford 1988, 1991), and a brief overview of Views is given.

Section 4 describes the plot-data interface. By adding appropriate generic functions to the plot-data interface, plot construction and behavior become effectively independent of the actual data set representation.

Section 5 describes a new, general algorithm for plot linking that exploits the plot-data relationship. In conventional linking of two scatterplots, the point symbols in each plot associated with a particular case will have the same color. Our linking scheme decides which point symbols in the two plots are to be linked by comparing their associated cases. Any (user-defined) test may be used to compare cases, not just the identity test. For a linking capability that is independent of the data set representation, we add a generic case comparison function to the plot-data interface.

2. PREVIOUS APPROACHES

In this section we outline previous approaches to the plot-data interface. We demonstrate that these approaches are inadequate for open-ended statistical software systems, especially those supporting high-interaction graphics.

In interactive plots, the dependence of plot on data persists beyond construction time. By interacting with a plot, the analyst can get information about the underlying data set. With a point-and-click style interface, for instance, the analyst identifies the case represented by a point symbol on the screen. In effect, the plot acts as a graphical interface to the data. Interactive plots are frequently dynamic, changing according to new information obtained from a data set: For example, the analyst can request a change of variables or that points representing outlier cases be colored red.

2.1 THE SIMPLE APPROACH

In the most common and simplest approach to the plot-data interface, each kind of plot can only display a particular type of data object. For reasons given later, this approach is unsatisfactory. Consider a bar chart that displays a vector whose elements are the bar heights. Note that:

1. The analyst is burdened with converting the data to such a vector prior to plotting. Keeping track of the proliferation of derived data sets and their interrelationships places an additional burden on the analyst.
2. This approach requires conversion functions for every combination of plot and data set type. It discourages the analyst from experimenting with new plotting techniques and data representations.
3. The data conversion often involves some loss of information. This conflicts with the notion that the plot acts as a graphical interface to the data. Suppose the bar chart displays the number of individuals in a data set participating in each of

five sporting activities. Because the plot actually displays a vector of frequencies extracted from the data set, it's not possible to select a bar and find out which cases it represents. Neither is it possible to change variables from sporting activities to say, racial group. The problem is that the connection between the plot and the original data set is lost.

2.2 An Object-Oriented Approach

Using object-oriented techniques, the *S* system (Becker, Cleveland, and Wilks 1988; Chambers and Hastie 1992) has improved on the simple approach. The plot functions in *S* are generic, which means they accept many different types of data objects as argument. For instance, the `plot` function can be invoked on either an array or a time-series object, and the appropriate plot is produced. The `plot` method for a type, `array` say, specifies the actions to occur when the generic function is invoked on an array object.

With generic functions there is no need for the analyst to preconvert the data to a particular format prior to plotting. The analyst is no longer burdened with producing and managing multiple data versions, and objection (1), stated previously, no longer applies. Essentially, each data representation requires a method instead of a conversion function, so objection (2) still applies. Adding a new data representation could demand new methods for every plotting function. Conversely, each new plot function needs methods for the data representations.

In principle, the generic plot function approach could overcome objection (3). Each method would have to build a plot that retained the association between the plot and the originating data set. The plot methods would be nontrivial, involving much duplication of effort, and would require detailed knowledge on the part of the implementor of both the plotting technique and the data set representation. Building a new kind of plot and/or data set representation would be an ordeal even for an experienced programmer.

At the other extreme there could be one method for a vector, for example, that actually builds a plot, while methods for other data types simply extract a vector and invoke the plot function on the vector. *S* plots lie closer to this extreme. (This is sufficient for *S* plots, which act as an interface to the data only in a limited sense.)

3. DATA AND PLOT MODELS

Our plot-data interface assumes a simple data model that is independent of the choice of data structure used to represent the data set. Our plots follow the Views model for statistical graphics (Hurley and Oldford 1988, 1991).

3.1 Data Model

We begin by introducing the `smoke-styles` data set, collected for an experiment conducted on smoking styles and described by Hand and Taylor (1987). We will refer to this data set in later sections. In this study, a subject smoked a cigarette and variables measuring the cigarette's pharmacological effect were recorded, as well as the subject's

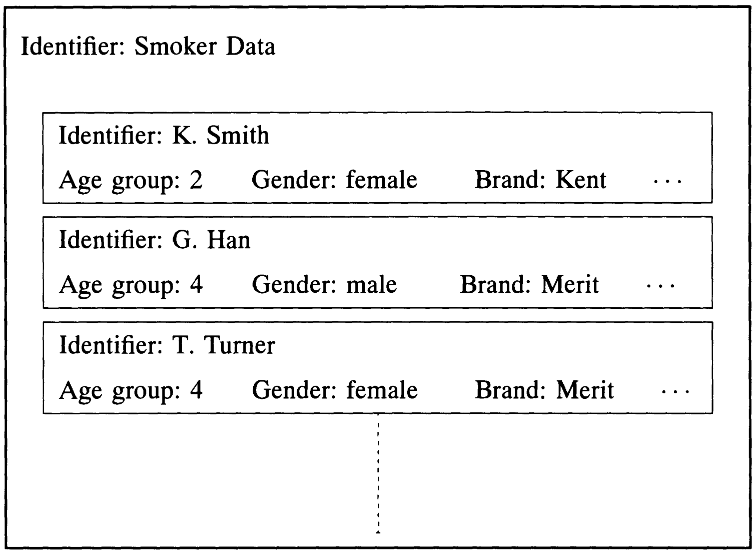

Identifier: Smoker Data
Identifier: K. Smith
Age group: 2 Gender: female Brand: Kent ···
Identifier: G. Han
Age group: 4 Gender: male Brand: Merit ···
Identifier: T. Turner
Age group: 4 Gender: female Brand: Merit ···

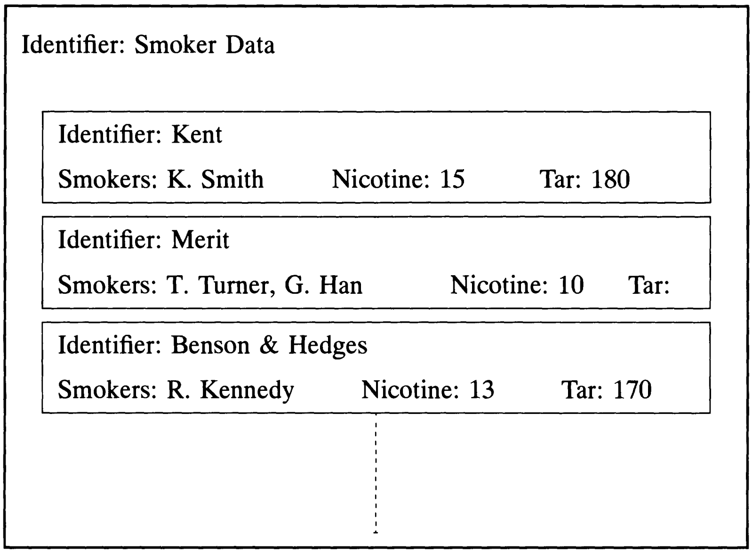
Figure 2. Model of `smoke-styles` *Data Set From the Smoker Perspective.*

age, gender, and cigarette brand. Additional variables were recorded for the cigarette brand. Figure 2 shows a model of the `smoke-styles` data set. (Smoker and brand names were not reported by Hand and Taylor [1987], so the values used here are artificial.)

The elements of our data model are cases, variables, and the data set itself. A case contains observations on an individual; for the `smoke-styles` data set, a case consists of all of the information collected that relates to a particular smoker. A variable is an index into the case used to extract an observation; in our example, the variables are `age group`, `gender`, and `blood nicotine`, among others. A data set, then, is composed of cases. We also regard an individual case as a data set. Additionally, a data set (and hence a case) may have an identifier, which describes or names the data set.

The data model assumes that information on a smoker and his/her choice of brand is accessible from a case. In some analyses, the cigarette brands rather than the smokers may be the primary focus of interest. Figure 3 shows a model of the data set from the cigarette-brand perspective; here each brand constitutes a case. The data model assumes that information on a brand, and the smokers selecting that brand, are accessible from the case that represents the brand.

Many different data structures could be used to represent the `smoke-styles` data set. A few possibilities are: The data set could be contained in a matrix in the conventional cases by variables format, where each smoker is a case. Then a row would represent a smoker, with entries for each variable associated with the smoker. An extra column could be used to store the case identifiers, which for this data would be the smoker's name. This data implementation is efficient for extracting information by smoker. It contains redundant information, however, because some smokers chose the same brand of cigarette. To save space, the cigarette variables could be stored in a separate data matrix. The matrices could be regarded collectively as a data set, where some cases are subjects and others are cigarette brands. Pointers from the smoker to the cigarette matrix would


Identifier: Smoker Data

Identifier: Kent
Smokers: K. Smith Nicotine: 15 Tar: 180

Identifier: Merit
Smokers: T. Turner, G. Han Nicotine: 10 Tar:

Identifier: Benson & Hedges
Smokers: R. Kennedy Nicotine: 13 Tar: 170

Figure 3. Model of `smoke-styles` *Data Set From the Brand Perspective.*

prevent any loss of efficiency in accessing all information (including cigarette variables) associated with a subject. Conversely, pointers from the cigarette to the subject matrix would allow for efficient access of information (including subject data) associated with a cigarette.

The purpose of our data model is to hide the actual structure used to represent the data set from the plotting software that displays the data set. Then, regardless of the data structure chosen, the same plots can be made. This allows a data representation to be chosen for efficiency reasons without regard to the demands of the plotting functions used to display the data.

3.2 Views Model

According to the Views model, statistical plots are collections of objects such as points, lines, labels, and axes. These objects are arranged in a hierarchy—a scatterplot consists of axes, a label, and a point cloud, which itself consists of point symbols. Similarly a scatterplot matrix consists of point clouds and labels, though arranged in a different format. An object appearing in the plot has an associated piece of statistical data—typically for the scatterplot it is the entire data set, for a point symbol it is a case, and for the point cloud it is the list of cases. The plot and each of its components are termed a *view*. A *simple view* is a view such as a point symbol or label that contains no other views.

A view is so-called because it provides a graphical representation of a piece of data, called the *viewed object*. A view contains a reference to its viewed object, and an image of a view is used as a graphical interface to the viewed object. Because a case is a kind of data set, a viewed object can be characterized as a data set or a list of data sets.

4. A PLOT-DATA INTERFACE

This section proposes a new plot-data interface. Our plot-data interface consists of generic functions, which the plot uses to obtain information from the data set. Thus plot construction and behavior become effectively independent of the data representation.

4.1 PLOT CONSTRUCTION

For a particular type of plot, we need to identify the information that the plot requires from the data set it views. We consider the information necessary for plot construction, and for concreteness we discuss the familiar scatterplot.

The following (Lisp) expression constructs a scatterplot of smokers in the data set `smoke-styles` with `puffed smoke` on the x axis and `blood nicotine` on the y axis.

```
(scatterplot :data smoke-styles
             :x "puffed smoke" :y "blood nicotine")
```

There are two essential stages where scatterplot construction requires information from the data set. First, cases (the smokers) must be extracted from the data set to become the viewed objects of the point symbols. For this purpose, the plot system uses a generic function called `list-cases`. This function takes a data set as argument and returns a list of its cases; for our example each case is a smoker. Second, coordinates must be extracted from the cases using the variables `puffed smoke` and `blood nicotine` as indexes into the case. For this the plot system uses a generic function `value-of`, which takes a case and index (the variable) as argument and returns a value.

For each data representation, the implementor of the graphics package assumes that appropriate methods for the generic functions `list-cases` and `value-of` exist. To use the graphics package, each new data representation then requires methods for the generic functions `list-cases` and `value-of`.

Data sets with more than one possible notion of "case" pose a small problem. For the `smoke-styles` data, we might wish to plot the data by cigarette (or by age group) rather than by individual smokers. The graphics system handles this possibility by allowing the user to supply a function to be used instead of `list-cases`. The following code constructs a scatterplot of cigarettes in the `smoke-styles` data set with (cigarette) variables:

```
(scatterplot :data smoke-styles :cases 'list-cigarettes
             :x "tar" :y "nicotine")
```

Suppose now we wish to build a plot of cigarettes that displays the subject variable `blood nicotine` on the x axis, where the x coordinate is the average of the `blood nicotine` values recorded for the subjects smoking that brand of cigarette. This could be achieved by supplying a function rather than a variable as the `:x` argument, as follows:

```
(scatterplot :data smoke-styles :cases 'list-cigarettes
             :x 'mean-blood-nicotine :y "nicotine")
```

Here the `mean-blood-nicotine` function will be applied to a cigarette object, instead of the default `value-of`.

Because data transformations are ubiquitous in data analysis, it seemed appropriate to add a data transformation utility to the Views system. Following Buja, Asimov, Hurley, and McDonald (1988), the Views system decomposes the transformation from n cases to n coordinates into stages as described in Hurley (1991). This implies that the value obtained from the case need not yield a single number because the Views system applies a series of transformations to the value to obtain the coordinate. Typically then, the argument to `:x` is only concerned with extracting information from cases, not with transformation. For example, the following is a plot of cigarette brands with the log of `nicotine` on the y axis and the mean of the logged `blood nicotine` values on the x axis:

```
(scatterplot :data smoke-styles :cases 'list-cigarettes
             :x "blood nicotine" :x-function '(mean log)
             :y "nicotine" :y-function 'log)
```

The advantages of the data transformation utility are clear: The components of the transformation can be quite simple, and the individual components can be changed independently of each other.

4.2 Plot Behavior

With interactive plots a user's actions cause the plots to respond and perhaps change. This behavior relies on a connection between a plot and its underlying data. Here we examine the information that interactive plots require from the data.

Recall that a view acts as a graphical interface to the associated viewed object. Using a point-and-click interface, the user first selects a view and then requests information on the viewed object via a button click or a menu selection. The user can then identify or inspect the viewed object. The identify operation prints out a short description of the viewed object. For each data set in the viewed object, `identify` prints the data set's identifier, obtained by applying the generic function `identifier-of` to the data set. Here we assume that `identifier-of` returns the data set's identifier, if present. If the data set has no identifier, the data set itself is printed. (Lisp's built-in `print` function is already generic.) Similarly, the inspect operation uses Lisp's built-in generic `inspect` function to obtain a detailed description of the viewed object. Of course, a data set implementor may choose to provide a specialized `inspect` method for the data representation.

Changing variables depends on obtaining information from the data set. To change variables, the user is offered a menu of variable choices. Therefore we need a generic `list-variables` function, that, given a data set, returns a list of its variables. Thereafter, coordinates are extracted from the cases as in plot construction.

We restrict changing cases to deactivating and later reactivating cases in the plot. Inactive cases are ignored by the plot: In a point cloud, a point symbol viewing an inactive case will not be drawn; deactivating a case for a histogram or fitted line causes the histogram bin counts or the fitted line's parameters to be recomputed.

For concreteness, consider deactivating a case in a point cloud. Using the graphical interface, the steps are as follows:

Step 1. Select a view whose viewed object is the case to be deactivated. The view could be a point symbol in a point cloud or a label in a display list showing the identifiers of the cases.

Step 2. Invoke the "deactivate case" operation on the point cloud. This deactivates the case in the point cloud that is "the same" as the viewed object of the view selected at Step 1. (In general, multiple views may be selected at Step 1, then invoking the "deactivate case" operation on a point cloud deactivates all of its cases that are "the same" as the viewed objects of the selected views.)

Step 2 requires a function (`eq-dataset`, say) that tests for "sameness" of data sets. Data set comparison could be done in several different ways: (1) by testing for object identity—the two data sets are the same object; (2) by comparing the contents of the data set; or (3) by comparing identifiers. Therefore, `eq-dataset` is a generic data set comparison function and different methods can allow for the different possibilities.

5. LINKING VIEWS

The basic idea behind scatterplot brushing, also known as painting, is that a point in a plot representing a case should have the same color as the point representing the same case in other plots (Becker, Cleveland, and Wilks 1987; McDonald 1982; Newton 1978). Such plots are often described as *linked.* As we will see, linking makes use of the plot-data interface.

The linking idea has been applied to objects other than point symbols (see, e.g., McDonald, Stuetzle, and Buja 1990) and used with drawing-style parameters (such as symbol shape or line thickness), not just color. Our linking algorithm also allows for these possibilities.

5.1 Approaches to Linking

We briefly describe some previous approaches to linking and, for simplicity, consider plots with point symbols only.

A simple linking algorithm (see, e.g., Tierney 1990) links the ith point symbol in one plot to the ith point symbol in another, essentially assuming that each point symbol views the ith case of a data set. This strategy is not appropriate when the two plots show different but overlapping subsets of cases, for instance.

A novel design feature of the Views model led to a simple though limited linking strategy (Hurley and Oldford 1991). Views was designed so that the same view object can be drawn at multiple locations on the screen and be a subview of more than one parent view. Consequently, two (or more) point clouds can have the same point symbol objects as subviews. Such point clouds are implicitly linked: Because the point clouds use the same actual point symbol object with its associated color to view a particular case, the same color will automatically be used to display the case in each of the point clouds. The limitations of this method are: (1) It requires identical point symbols in the plots, not just the colors and (2) it does not permit linking of different types of plots, such as a scatterplot and histogram.

A more general strategy used by McDonald (1986) and Stuetzle (1987) associates color information with the cases rather than with the point symbols, and then constrains the point symbol to appear with the color of its case. When multiple point symbols view the same case, they are implicitly linked. This scheme is conceptually simple, but we reject as artificial the notion that a case has a color attribute. Every data set implementation would have to deal with color attributes—this is impractical and inconsistent with our goal of independent plot and data set modules.

In our proposed linking strategy, the Views system keeps track of the color and linking information. The algorithm determines which point symbols are to be linked by comparing their associated cases, using a generic data set comparison function. Therefore, our linking scheme makes use of the plot-data interface.

5.2 Drawing Styles

Prior to describing the linking algorithm, we present some necessary background on drawing styles for views.

A view is drawn by recursively drawing each subview, actual drawing occurring when the simple views are drawn. Each simple view has a drawing style, consisting of all parameters controlling how the view is to be drawn. Minimally, drawing-style parameters allow a view to be invisible, highlighted, or to take on different colors. In addition, point symbols have shape and size and may be outlined or solid; labels have font information.

Point symbols and labels are examples of *single-style* views, where the view has just one highlight value, color, and so on. Other views, such as bars and lines, are *multistyle*, and can have multiple colors.

A bar is a simple view drawn as a rectangle and used as a component of a bar chart, histogram, or boxplot. Normally, the viewed object of a bar is a list of one or more cases. (Because the data sets viewed by simple views are typically cases, we will refer to them as such without further qualification). With drawing styles, a bar can be drawn filled or not, in any color. The bar may be partitioned horizontally or vertically into rectangular segments, where each segment has a different combination of drawing-style values. A segment represents a subset of the bar's cases, with the segment size in proportion to the number of cases in the subset. (Alternatively, one could envisage constructing a bar object for each partition element, but this is inefficient for dynamic applications like brushing, where the partitions are frequently modified.)

Multistyle views, such as a bar, usually have drawing-style values corresponding to each of its cases. A pie and sunflower (with one ray for each case, or alternatively, one ray for each combination of style values) are other examples of multistyle views.

The generic function `set-drawing-style` changes the drawing style of a view. This function is invoked by the user (using a command-style or graphical interface) or automatically by the system when the style of a linked view is altered. For example, the following sets the color of the view `foo` to red:

```
(set-drawing-style foo :color red-color)
```

More precisely, this sets all of `foo`'s color style values to red and redraws `foo`. We can also set the color of the part of `foo` corresponding to a particular case, `smoker-1` say:

```
(set-drawing-style foo :color red-color :element smoker-1)
```

If `smoker-1` does not correspond to any of the cases viewed by `foo`, the view is unchanged. The generic data set comparison function from Subsection 4.2, `eq-data set`, is used to determine correspondence among cases.

5.3 LINKING ALGORITHM

From the discussion on drawing styles we note that linking need only be considered for simple views. In fact, we designate a subset of simple views as *linkable*.

For speed, a linked view, `foo` say, has a list of the views to which it is linked. When a drawing-style value of `foo` is changed, the new style is automatically passed to its linked views, using `set-drawing-style`. The case or cases corresponding to the changed style must also be passed to the linked views. Like the case, if the style is not relevant for a linked view (for example, a point symbol has a shape style, a bar does not), it is ignored and the view unchanged.

Next we describe how the links are built. A *link table* consisting of all views currently linked is maintained. There are three steps involved in linking a view `foo`. Assume `foo` is linkable and as yet unlinked:

Step 1. For the first view, `view-1` say, in the table, compare its cases to those of `foo` using the generic function `eq-data set`. If there is a match, add `foo` to `view-1`'s list of links and, conversely, add `view-1` to `foo`'s list of links. Repeat this for every view in the link table.

Step 2. Add `foo` to the link table. Note that this step is performed even if there are no matches in Step 1, ensuring that sequentially linking views is not order dependent.

Step 3. Update the styles in `foo` to be consistent with those of its links and redraw.

When the view `foo` to be linked is not simple, then the previous process is performed for each linkable descendant of `foo`—that is, each linkable view in the `foo` hierarchy.

We note that:

- The linking procedure is the same for all linkable views regardless of both its type and the type of containing plot (if any). Therefore the procedure will work for new views to be designed in the future.
- The algorithm allows subviews of a view to be linked to each other. This is useful because, for instance, one could construct a view containing multiple scatterplots and histograms that are linked to one another.
- There may be multiple link tables, one for each group of related views. (As yet, each view can belong to at most one link table.)
- There are also functions for unlinking a view and for clearing the entire link table.

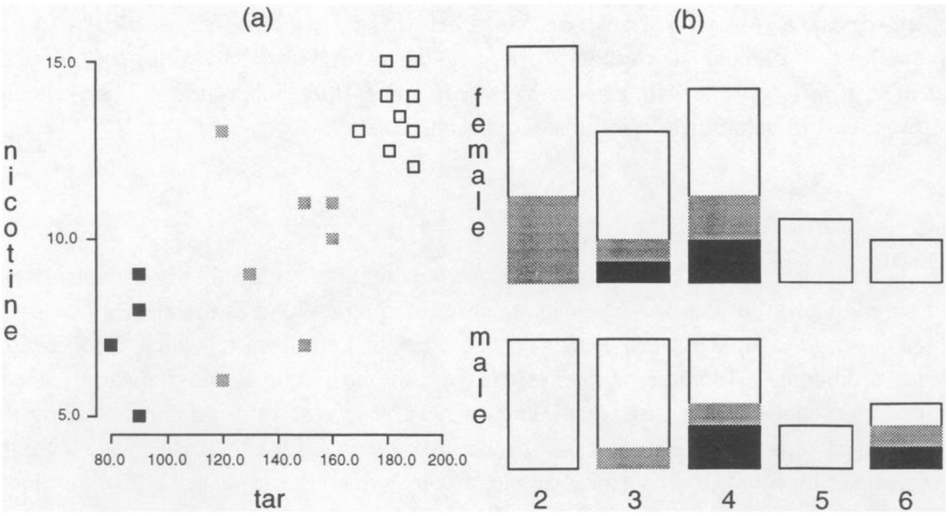


Figure 4. Linked Plots of Smoker Data. (a) Cigarette brands and (b) smoker age by gender.

5.4 Data Set Comparison

The generic data set comparison function `eq-dataset` plays an important role in linking, determining which views can be linked. In Subsection 4.2 we described `eq-dataset` as comparing two data sets to see if they are "the same" in some sense. Here we consider other possibilities.

As motivation, consider the `smoke-styles` data. As remarked in Subsection 3.1, here one can regard either the smokers or the cigarette brands as cases. In Figure 4a we see `nicotine` and `tar` plotted for the cigarettes, and in Figure 4b we see a bar chart of the smokers' ages, grouped by gender. (The age variable is categorical, with age "2" recorded for smokers in their twenties, and so on.) We wish to answer such questions as: (1) What is the tar content of the cigarettes smoked by young females? (2) Which group tends to smoke low-tar cigarettes?

To answer these questions, we require links between a cigarette brand point symbol and each bar viewing a smoker of that brand. This implies that the data set comparison function should return true for a smoker-cigarette pair when the smoker chooses that brand of cigarette. Let us call this more general data set comparison function `data set-intersection`. For any two data sets this function returns true if they have some data in common. Using `data set-intersection` in place of `eq-dataset`, question (1) is answered by highlighting the bar viewing the females in their twenties, causing the point symbols viewing the brands this smoker group chooses to become highlighted. Similarly, question (2) is answered by highlighting the low-tar points representing the cigarettes, so that the barchart shows the frequency of selection of low-tar brands by age and gender. From Figure 4 we see that, for the smokers in the study, young smokers do not select low-tar brands, and the older women (in their fifties or beyond) smoke high-tar brands exclusively.

The data set comparison function determines which views can be linked. We note

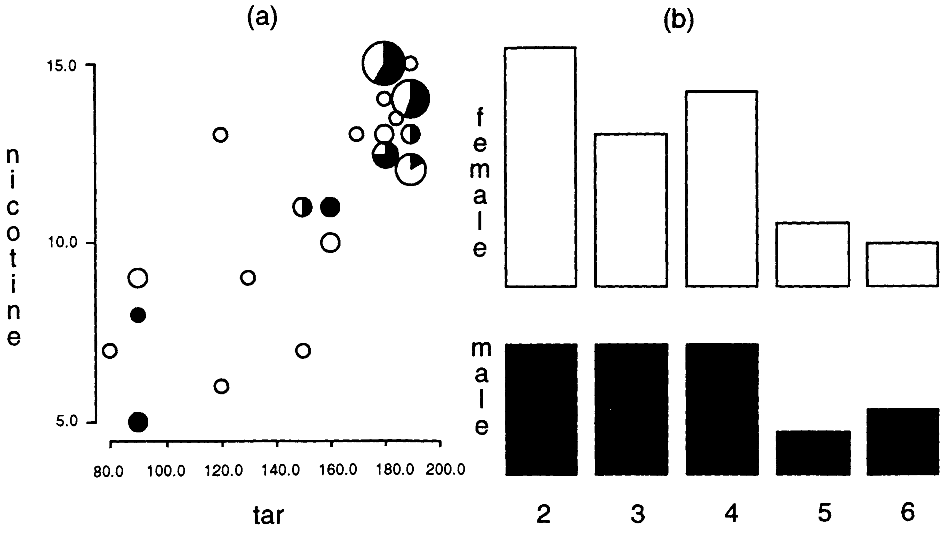


Figure 5. Linked Plots of Smoker Data. (a) Cigarette brands and (b) smoker age by gender.

that:

1. Views representing cases of different types can be linked as long as the data set comparison function has an appropriate method for each pair of case types.
2. So far in our discussion we have assumed symmetric links; if `view-1` is linked to `view-2`, then `view-2` is linked to `view-1`. The possibility of asymmetric links is interesting, however, and could be implemented by modifying Step 1 of the linking algorithm (p. 375). This calls for an asymmetric data set comparison function, perhaps `dataset<=`.

5.5 MULTISTYLE VIEWS

For Figure 4a, the choice of the single-style point symbol view to represent each cigarette brand is perhaps not ideal. Suppose we color the point symbols viewing smokers white for female and black for male. This will also partition the point symbols viewing cigarettes into black and white, but the partition is not unique. A cigarette smoked by both a male and female could appear as either black or white, depending on which color was more recently assigned.

Therefore, we allow the point symbol used by the scatterplot to be replaced by a multistyle view, such as a bar or pie, as shown in Figure 5a. This brings up the question, "How many sets of style values does a multistyle view have?" Even though the pies in Figure 5a each view a single case that is a cigarette brand, each pie has as many sets of style values as there are smokers who selected the brand. Also, each pie has area in proportion to the number of smokers. We notice that while high-tar cigarettes are the most popular, there does not appear to be a gender difference in the choice of brands as distinguished by their tar and nicotine content.

In general, a multistyle view uses the `list-data-elements` generic function from the plot-data interface to decide how many sets of style values are to be used. This function is applied to the cases in the viewed object and returns a list of the data elements contained in the cases. Then the multistyle view has one set of style values per data element. Normally the data elements are simply the cases themselves. For hierarchical cases such as a cigarette brand containing smokers, however, the data elements can be the "smallest" cases in the viewed object—the smokers, in our setting.

6. CONCLUSION

A plot-data interface provides a way of managing the complexity caused by the multiplicity of plot types and data set representations in statistical programming environments. This article described the primary elements of such a plot-data interface for highly interactive, linkable statistical plots.

The interface relies on the programming techniques of data abstraction and generic functions to hide the actual data set representation used from the plot displaying the data set. The plot is then effectively independent of the actual data representation. The same strategy may be used to deal with the dependence of model-fitting procedures on data.

The ideas described in this article have been implemented by the author in Common Lisp and CLOS (Keene 1988; Steele 1990), as part of the Quail environment for statistical computing. A beta release of the software is available from the Statistical Computing Laboratory at the University of Waterloo, in ftp directory `pub/Quail` on `setosa.waterloo.ca`. While our approach to statistical plots, and the abstraction barrier between plot and data, could of course be implemented in languages other than Lisp, we recognize that our approach is inherently object-oriented and thus an object-oriented implementation language will be a definite advantage.

[*Received January 1992. Revised June 1993.*]

ACKNOWLEDGMENTS

The author thanks R. W. Oldford for helpful discussions.

REFERENCES

Abelson, H., Sussman, G., and Sussman, J. (1985), *Structure and Interpretation of Computer Programs*, Cambridge, MA: MIT Press.

Becker, R. A., Cleveland, W. S., and Wilks, A. R. (1987), "Dynamic Graphics for Data Analysis," *Statistical Science,* 2, 355-395.

Becker, R. A., Chambers, J. M., and Wilks, A. R. (1988), *The New S Language*, Pacific Grove, CA: Wadsworth and Brooks/Cole.

Buja, A., Asimov, D. A., Hurley, C., and McDonald, J. A. (1988), "Elements of a Viewing Pipeline for Data Analysis," in *Dynamic Graphics for Statistics*, eds. W. S. Cleveland and M. E. McGill, Pacific Grove, CA: Wadsworth and Brooks/Cole.

Chambers, J. M., and Hastie, T. J. (eds.) (1992), *Statistical Models in S,* Pacific Grove, CA: Wadsworth and Brooks/Cole.

Hand, D. J., and Taylor, C. C. (1987), *Multivariate Analysis of Variance and Repeated Measures*, New York: Chapman and Hall.

Hurley, C., and Oldford, R. W. (1988), "Plots as Hierarchichal Views of Statistical Objects," 19-minute video (VHS format), STAT-22-88, Statistics Technical Report Series, University of Waterloo, Waterloo, Ontario

——— (1991), "A Software Model for Statistical Graphics," in *Computing and Graphics in Statistics IMA Volumes in Mathematics and its Applications*, (vol. 36), eds. A. Buja and P. Tukey, New York: Springer-Verlag.

Hurley, C. (1991), "Some Interface Issues for Interactive Statistical Graphics," in *Computer Science and Statistics: Proceedings of the 23rd Symposium on the Interface,* Fairfax Station, VA: Interface Foundation of North America.

Keene, S. E. (1988), *Object-Oriented Programming in Common Lisp*, Reading, MA: Addison–Wesley.

McDonald, J. A. (1982), "Interactive Graphics for Data Analysis," Ph.D. dissertation, Stanford University, Statistics Department.

——— (1986), "Antelope: Data Analysis with Object-Oriented Programming and Constraints," in *Proceedings of the American Statistical Association, Section on Statistical Computing,* 1–10.

McDonald, J. A., Stuetzle, W., and Buja, A. (1990), "Painting Multiple Views of Complex Objects," *SIGPLAN Notices*, 25, 245–257, Proceedings OOPSLA/ECOOP '90.

Newton, C. M., (1978), "Graphics: From Alpha to Omega in Data Analysis," in *Graphical Representation of Multivariate Data*, ed. P. C. C. Wang, New York: Academic Press.

Steele, G. L. (1990), *Common Lisp, The Language* (2nd ed.), Bedford, MA: Digital Press.

Stuetzle, W. (1987), "Plot Windows," *Journal of the American Statistical Association*, 82, 466–475.

Tierney, L. (1990), *LISP-STAT An Object-Oriented Environment for Statistical Computing and Data Analysis*, New York: Wiley Interscience.