Gossip:
On The Architecture of SpreadPlots

Forrest W. Young, Pedro Valero-Mora,
Richard A. Faldowski & Carla Bann[1]

1. Author Footnote: Forrest W. Young is Professor Emeritus at the L.L. Thurstone Psychometric Laborattory. University of North Carolina at Chapel Hill. CB 3270 DA, Chapel Hill NC., USA 27599-3270.(e-mail: forrest@unc.edu). Pedro Valero-Mora is assistant professor of Data Processing at the University of Valencia. Departamento de Metodología de las Ciencias del Comportamiento. Facultad de Psicología. Av. Blasco Ibáñez, 21. Valencia. CP: 46010. Spain. (e-mail: valerop@uv.es). Richard A. Faldowski is assistant professor in the Department of Psychiatry and Behavioral Sciences at Medical University of South Carolina; 135 Rutledge Avenue, Room 1201; P.O. Box 250550; Charleston, SC. 29425 USA; (e-mail: faldowra@musc.edu). Carla M. Bann is a quantitative psychologist at Research Triangle Institute, P.O. Box 12194, Research Triangle Park, NC, USA 27709-2194. (e-mail: cmb@rti.org).

September 6, 2001

TGossip: On The Architecture of SpreadPlots

Forrest W. Young, Pedro Valero-Mora,
Richard A. Faldowski & Carla Bann

Abstract
A spreadplot is a visualization which simultaneously shows several different views of a dataset or model. The individual views can be dynamic, can support high-interaction direct manipulation, and can be algebraically linked with each other, possibly via an underlying statistical model. Thus, when a data analyst changes the information shown in one view of a statistical model, the changes can be processed by a the model and instantly represented in the other views. Spreadplots simplify the analyst's task when many different plots are relevant to the analysis at hand, as is the case in regression analysis, where there are many plots that can be used for model building and diagnosis. On the other hand, the development of a visualization involving many dynamic, highly-interactive, directly manipulable graphics is not a trivial task. In this paper we discuss a software architecture which simplifies the spreadplot developer's task. The architecture addresses the two main problems in constructing a spreadplot, simplifying the layout of the plots and structuring the communication between them.

Keywords: Dynamic graphics, statistical programming, layout, coordination

1 Introduction

Computational power has increased to such an extent that it is now possible, even on the typical computer purchased for use at home, to quickly estimate complex statistical models and provide graphical representations of them. All this power, however, has not been problem-free. Perhaps the main problem is the burden placed on the user by complex user interfaces with little or no structure. Such interfaces can provide the user with a myriad of windows, scroll bars, menus, buttons, dialog boxes, etc. If these are not structured in some way the user quickly becomes overwhelmed. At the best, with such systems the user spends much time and effort managing the interface, at the worst these actions prevent the user from focusing on the task at hand. The user can become mentally exhausted just tracking the consequences of making changes in the interface, with no cognitive power left for tracking the consequences of making changes in the statistical analysis.

Take, as an example, the plots that are usually recommended when using a simple linear model. Just the plots for raw data can be overwhelming, since these include histograms, frequency polygons, dot plots, box plots, quantile plots, normal-probability plots, quantile-quantile plots, scatterplots, parallel-coordinate plots, mosaic plots, spin plots, scatterplot matrices, and even tour plots (each of which has several variants). Then there are the plots for the analysis of the data. Considering just linear regression, the recommended plots include regression plots, fit plots, added variable plots, leverage plots, influence plots, transformation plots, weight plots, and more. Statistical packages that do not offer a well-devised way of interacting with all of them discourages the use of plots. Textual output is simply much more convenient in comparison. On the other hand, properly presented, these plots can be very informative and can lead to great insight into the nature of the data.

Insight/Effort: Consider the concept we refer to as **I/E**, the user’s **Insight/Effort** ratio, an index that summarizes the situation previously described. A computer program implementing statistical plots should try to maximize this index: It should keep the insight high while keeping the effort low. In the example about the linear model, if there were a way to present many of these plots simultaneously, without the user having to spend a lot of time rearranging plots, we could increase the user’s insight, while minimizing the effort required to gain the insight. Optimizing this insight/effort ratio becomes still more crucial when you realize that the data analyst generally considers several models before coming to a conclusion, and that the process of data analysis is a cyclic process that takes place over long periods of time and many sessions with the statistical system.

We propose a solution to this problem, and investigate the consequences of our solution. Our solution is based on the observation that in many, indeed perhaps in most data analysis situations, the plots for a given problem are all very inter-related with each other. Our solution is a way to create, interact and remove many related plots as though they were a single unit.

Specifically, we propose an architecture for dealing with sets of plots, or, more generally, multiple, algebraically linked, highly dynamic, interactive views of a dataset or of a model of a dataset. We refer to the architecture as a spreadplot. The spreadplot architecture provides a method for implementing such multi-view systems in a coherent and relatively simple way. In particular, spreadplots help to deal with two of the most essential problems with multi-view systems for data visualization: Communication between views, so they can act as coordinated sets, and Layout, so that they can be arranged on the screen without much effort by the user.

The Gossip: An informal way of looking at the workings of our spreadplot architecture is to regard it as a village with a gossip. Like all villages everywhere, the village has its’ physical layout, and it has its’ communication structure. And, like everywhere, the villagers don’t communicate with each other directly, rather, they “gossip”.

Gossip involves a centralized “gossiper”. The gossiper listens. It listens, not only to those it already knows, but to anyone who is willing to talk. When it hears something, it doesn't stop to think, it just tells everyone else about it. It does pay attention, however, when someone says “I don't want to hear about that anymore”, and never talks about that topic with that villager again. Other than that, it doesn't do anything except pass the gossip on to all those that don't say they don't want to listen. It never wastes any time repeating itself, nor in talking to those that don't want to hear. Of course it goes without saying that the villagers never talk directly with each other, only to the gossiper.

Thus, we call our SpreadPlots architecture the “gossip architecture”. Messaging is managed by a centralized manager. The individual objects don't communicate directly with each other, only with the manager.The manager receives messages from any object, taking new objects into its group, no questions asked. It then repeated the message so that everyone can hear. But, while the manager is indiscriminate about adding new members to its gossip-group, it is very concerned about not sending unwanted gossip to the members. Thus, the manager doesn't send messages to those that say they can't use them. Finally, the manager is also responsible for layout, although the most important role for the manager is that of message manager.

In this paper we describe how the problems of communication and layout have been managed in the context of currently existent tools for data visualization. We then introduce the fundamentals of spreadplot architecture, and describe how these fundamentals deal with communication and layout. We then present an example which illustrates both layout and communication. We close the paper with a discussion of the consequences of spreadplot architecture.

2 Overview of Previous Multiple Plot Systems

In this section we review previous developments concerning multiple plot systems, with emphasis on how the two main issues of communication and layout have been treated.

2.1 Communication

The canonical example of communication between plots is what Wills (1999) calls the linking data paradigm. In this situation, different data views are linked by its cases. It works in the following manner: simple data views are created and linked such that interacting with a view updates other views to reflect their mutual relationships at the case level. For example, moving the cursor over points in a scatterplot may highlight corresponding points in another scatterplot, a technique termed the brushing technique by Becker, Cleveland and Wills (1987). Brushing is implemented in many statistical systems. See, for example, XGobi (Swayne, Cook and Buja, 1998).

Even in this basic data linking paradigm two approaches can be distinguished (Tierney, 1990). In the *common data model,* linked plots are different views of the same data. Thus, all changes to the data set should be shown in all views, as in the Plot windows system (Stuetzle, 1987). In the *separate data model,* on the other hand, plots are separate entities containing objects that are indexed in some coherent way. Plots that are linked will propagate the interactions on their entities to the entities on other plots that have the same index value. XlispStat (Tierney, 1990), DataDesk (Velleman, 1992) and S (Becker and Chambers, 1984; Becker, Chambers and Wills, 1988) are tools that incorporate a separate data model as default. Tierney (1990) provides code for extending XLispStat in order to implement the common data model. Each model has its advantages and disadvantages. For example, with the common data model, as each observation has its own identifier, it is quite easy to link a plot of a section of the data to a plot of all the cases of the dataset in it. On the other hand, this model would force the linkage of all features in plots, while with the separate data model an observation might have a different feature in different plots.

The concept of this basic data linking paradigm has been extended in several directions. For example, Wills (1999) signals that it would be very useful to link cases with ones in other data matrix (for example when a data matrix is the result of an aggregation of other data matrices and each case in one stands for several cases in the other) or to link a plot of data with a model of the data. A program that incorporates this last type of link between data and model is Datadesk (Velleman, 1992). Wilhelm (1999) attempts to classify all the possible extensions to linking into the following three categories: linking sample populations, model linking and graphical linking. Linking sample populations includes the following cases: identity linking (cases or observations at the same level are linked), hierarchical linking (selecting an object at a higher level selects all the related objects at a lower level) and distance and neighborhood linking (selecting an object selects objects that are related or close to it). Model linking includes the case where the same objects are represented in different plots (like for example when a variable is present in two different plots that reveal different properties) and when scale objects are linked (like when a slider represents a set of possible parameters for a model that can be controlled interactively to find appropriate transformations). Finally, graphical linking refers to when the axes or the frames of the plots are linked to allow for making valid comparisons.

Coordination of multiple views for information visualization is a topic that has recently received much attention from researchers on Human Computer Interaction (HCI). The reason of this interest stems from the observation that developers of software often need to include different sources of data in the interface in order to help the user to take the right decisions. Wang Baldonado et al. (2000) describe an example of a multimedia system to support the task of identifying seizures in infants. These seizures are very subtle events and doctors must simultaneously review of physiological data and visual observations of the infants's movements. These authors affirm that designers of multiple view systems make many unnecessary design mistakes, introducing unnecessary complexities and inconsistencies when trying to coordinate different views in a interface. They provide guidelines to help designers to avoid these mistakes.

North & Shneiderman (2000) present a system for coordinating multiple views. This system allows for taking advantage of simple relationships between data such that coordination between views is possible without programming. This objective would differ from our objective in this paper because we are interested in the exploration of innovative coordinations between the different views, and, even though we try to make this as easier as possible, a certain amount of programming is expected.

In summary, the continued development by the variety of researchers mentioned above, suggests that complex linking between multiple views would appear to be potentially useful in the statistical programming situation. Furthermore, flexible linkage of multiple views appears to be crucial for the development of successful tools for visualizing complex data. Indeed, it would be best if such tools were so simple, flexible, and powerful that the statistical system user could apply them to his or her own data.

Among the various systems, S and Lisp-Stat seem to be best suited for research and development of methods for complex linking. We have, in fact, been using Lisp-Stat for such research for a number of years, and it is indeed a very powerful and flexible tool for this purpose. However, our several attempts to develop multiple view visualizations have taught us that it is very difficult to attain sound implementations without the proper software architecture. Moreover, since the views are often interconnected in a very subtle net of links, it is very difficult to add, modify or remove new elements without breaking the entire structure. The architecture we will describe has the advantage of being able of managing the communications smoothly and of permitting modifications. Consequently, the developer will feel more confident of experimenting with features for interconnecting plots.

2.2 Layout

How to arrange plots in the display is a very important issue for visualization programs (Murrell, 1999).Indeed, there are some plots, like scatterplot matrixes (Chambers et al., 1983) or trellis displays (Becker, Cleveland and Shyu, 1996), that are formed just by arranging simpler plots according to certain rules. Scatterplot matrices, for example, arrange scatterplots side by side so each variable in a dataset is graphed against the others variables, with the graphs being displayed as a row or a column of the matrix. This allows for rapid inspecting of all the bivariate relationships among variables, permitting the detection of outliers, nonlinearities, and other features of the data.

There are some computer tools for visualization of statistical data that already incorporate features for managing and creating multiple views. However, many other programs create plots as separate entities than can not be arranged relative to each other afterwards. Some programs include the capability of empirical linking but programming additional behavior is not possible. Arc (Cook & Weisberg, 1999) for example provides a great selection of plots related with regression analysis, but, even though the plots referred to a dataset are linked, there is no further interaction among them.

A tool that includes a feature for setting customized layouts is Datadesk (Velleman, 1992). This feature is based on a special type of window called a corkboard. Corkboards allow other windows (which could provides views of data or models) to be pasted into them. These views can be related with a statistical problem that the analyst may find interesting. These models and plots are empty of data, but methods are provided so that users can carry out their own analyses. This allows the user to develop customized statistical procedures or specialized applications.

Datadesk goes one more step: It provides buttons that can be programmed using a menu-oriented scripting language. Afterwards, the buttons can be pasted into the corkboard, so that actions can be taken from them. This lets the user develop innovative analysis or visualization capabilities that step beyond the already considerable capabilities of Datadesk. Other features of Datadesk are derived variables and formulae that are recomputed when the terms involved in them change, so that linking between models and plots of the results are very natural. Of course, empirical linking of the cases in plots and models is also available.

Datadesk is certainly a very flexible and useful program for interactive graphics, providing a very good choice of options that are probably sufficient for the majority of users. However, it has some limitations. For example, plots in corkboards are set manually in a specific layout and in a pre-specified number. Therefore, if the input consists of a variable number of plots or windows, like it happens with scatterplot matrixes or trellis displays, the corkboard will not be able to accommodate them automatically. Also, new connections between views can not be explored because the programming language of Datadesk can be executed only from the buttons.

3 Introduction to Spreadplots

This section will introduce spreadplots, a group of algebraically linked dynamic and highly interactive plots. We provide a brief overview of the history of spreadplots, indicating the contribution of this paper relative to previous papers on the subject. We also present an example from the point of view of the data analyst who is using a spreadplot as part of a statistical data analysis task.

3.1 Background

Young (Young, Faldowsky and McFarlane, 1992; Young, Faldowsky and McFarlane, 1993; Young and Bann, 1997) has developed ViSta, a statistical system based on Lisp-Stat. Lisp-Stat is a statistical programming environment developed by Luke Tierney (Tierney, 1990) that features a object-oriented approach for statistical computing and that allows for interactive and dynamic graphs. In particular, Tierney extended the Lisp language to support vectorized arithmetic, basic statistical computations, a window system and tools for building graphics, with special emphasis on dynamic graphics. ViSta incorporates the object-oriented approach as part of its internal and external functioning. In particular, it extends Lisp-Stat with additional graphical, statistical and data objects; it provides objects for mapping the process of data analyses and it has objects that guide novices through their early attempts to carry out analyses. All these characteristics shape a system that has been shown to be appropriate for students and teachers of statistics, for researchers who wish to explore and analyze their data, and for developers of computational and graphical statistics.

An innovative aspect of ViSta is the spreadplot, a group of several plots that simultaneously provide alternative views of data or model objects (Young, Faldowsky and McFarlane, 1993). The plots in a spreadplot are linked so that changes in one plot are reflected in other plots. The plots can be dynamic, using animation to convey meanings that are not easily visualized by static plots. They also allow the user to interact with the plot to create and control its dynamic aspect. These characteristics mean that users can use spreadplots to explore the data or models in a more detailed manner. We will introduce in the following section a description of spreadplots with an example to motivate and clarify the discussion of the specific issues of communication and layout which follows.

A spreadplot is the graphical equivalent of a spreadsheet. In spreadsheets the data are contained in cells which are arranged in tables, just like values in a matrix of data. The cells are linked by formulae, so that when the spreadsheet user makes a change in one cell, the algebraic links cause the linked cells to change correspondingly, enabling the user to easily explore the ramifications of the change. In spreadplots, the cells contain graphs or tables, rather than numbers: The datasheet cells become plot-cells. User interaction with any of the plot-cells causes linked plot-cells to change accordingly. Each plot-cell can represent a part of the statistical problem in hand. This setting allows the user to explore the relationships between the several plot-cells in a spreadplot much like you would do with cells in a spreadsheet.

3.2 Example

Spreadsheets are a natural environment in accountancy. Algebraic computations produce summaries of special relevance for the analyst. Special arrangements of computed cells create a representation of an algebraic problem that can then be submitted to "what if" analysis. For example, various tax deduction rules can be represented by formulas linking relevant cells of a spreadsheet. The accountant, using his knowledge of tax regulations, can then play with the raw values in order to get a more profound understanding of the consequences of different actions with regard to tax payment.

Spreadplots are designed to be a provide a similar environment for statistics. Sets of algebraically linked plots are used to represent a statistical problem. For example, the normal distribution is commonly assumed to underlie much of what is done in statistics. Thus, a common task facing the data analyst is to check on this assumption. Since one implication of this assumption is that a group of normally distributed variables will have linear bivariate relationships, researchers are commonly told they should plot the bivariate relationships to see if they appear to be linear, and to transform the variables when the raw variables are not linearly related.

Figure 1 shows a spreadplot which displays the bivariate relationships between five variables. This spreadplot also provides a mechanism for transforming the variables to make the bivariate plots look more linear. These bivariate relationships are shown in the scatterplot matrix occupying the left portion of the display. At any given time, one of the bivariate relationships is the “focal relationship” of the spreadplot, and one of the variables in this relationship is the “focal variable”. The task of the data analyst is to transform the focal variable so that the focal relationship appears to be linear.

The analyst determines which bivariate relationship is the focus by clicking on one of the small scatterplots in the scatterplot matrix. As a consequence of the click, the focal relationship shifts to the clicked scatterplot, which is shown in the bivariate scatterplot in the lower-right portion of the spreadplot. The user’s click also makes the horizontal variable in the bivariate scatterplot becomes the “focal variable” of the spreadplot. The normal-probability plot of the focal variable is shown in the upper-right portion of the spreadplot.

Note that when the user clicks a diagonal cell of the scatterplot matrix, the focus of the graphical analysis switches to the variable shown in the clicked cell, with the individual scatterplot displaying the transformation of the variable in effect at the moment (i.e., the scatterplot shows the untransformed values of the focal variable versus the transformed values of the variable).

At any given moment, the goal of the analyst is to make both the bivariate scatterplot and the normal-probability plot as linear as possible (since the normal-probability plot is linear when the variable is normally distributed). The overall goal is to make these two plots as linear as possible for every variable.

The mechanism for transforming variables to remove nonlinearities is based on the family of scaled power transformations (Box & Cox, 1964; Emerson 1991; Tierney 1990; Cook and Weisberg, 1999). These transformations are defined as

$$f(y) = \begin{cases} (y^p - 1)/p & \text{for } p \neq 0 \\ \log(y) & \text{for } p = 0 \end{cases} . \qquad \textbf{(EQ 1)}$$

Manipulating the slider shown in Figure 1 modifies the value of p in the equation, and then applies the transformation formula to the current variable using the p-value shown on the slider. Transformations with p>1 make left-skewed distributions more symmetric and transformations with p<1 have the same

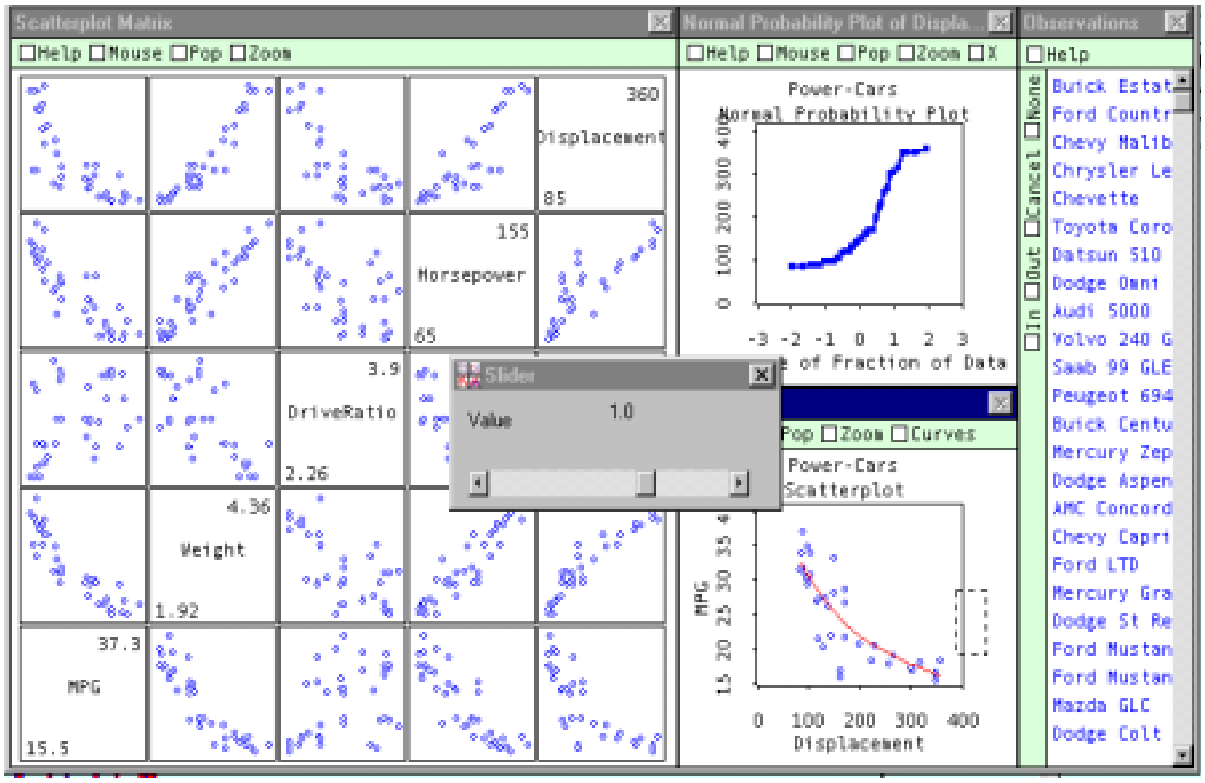

Figure 1: Spreadplot for the Box-Cox Transformation

effect with right-skewed distributions. The normal probability plot allows for assessing this effect. Additionally, the matrix of scatterplots and the focal scatterplot also change when the transformation is applied, so that linearity between pairs of variables can be evaluated. The focal scatterplot also can has several tools for helping with this evaluation, including least squares lines, loess smoothers, kernel smoothers, etc. Just for illustration purposes a lowess smoother with a parameter of 0.7 is shown in the scatterplot in figure 1.

Notice the coordination between plots. Both the slider and the scatterplot matrix react to the user's interactions. Specifically, when a cell of the scatterplot matrix is clicked, a message is sent telling the normal-probability plot and the scatterplot to change themselves to show the new focus. Also, when the slider if moved, a message is sent telling all of the plots to change themselves in light of the new value of p in the equation above.

These messages cause changes to take place according to clearly defined rules. The scatterplot, for example, evaluates whether the cell in the scatterplot matrix that has been clicked is in the diagonal. If the result of the evaluation is true, a transformation plot is shown. If not true, a scatterplot is shown. When the slider is moved, the plots are modified according to the rule instantiated in the equation above. Additionally, the points in the plots are linked, so the modification of their properties (color, symbol, selection, etc.) propagates to other plots.

This rich interaction introduces several complexities that makes the programming of spreadplots somewhat complicated. In our early implementations of spreadplots it became very clear that controlling the flow of messaging between plots can be complicated and delicate, and that a clearly designed and well thought-out software architecture was needed to avoid horrendously complex and inefficient messaging. Indeed, in our early work, the seemingly simplest approach was taken: Each plot sent messages directly to other plots. However, after some experimentation with a prototype of a spreadplot, we often would want to modify it by adding or removing a view from it or by changing the interactions with the user. Nevertheless, such changes all too frequently would break the messaging system. As a consequence, this experience has lead us to our current (the sixth, and we believe, final) architecture. This architecture is described in the next section.

4 The Architecture of Spreadplots

This section of the paper focuses on the architecture that we have developed which underlies the two most important aspects of spreadplots: layout and messaging. Of these two, message passing is preeminent because it is through message passing that the cells of a spreadplot are connected, and connection is the most innovative and beneficial aspect of spreadplots -- their "raison d'etre". Our approach to layout, while perhaps of less importance, is quite flexible in comparison to other systems, permitting the spreadplot designer wide latitude in the appearance of the spreadplot. We describe these two systems in this section.

4.1 Gossip Messaging Architecture

Messages can be created by a user by typing commands (as above) or indirectly as a consequence of interactions with the user interface (i.e. pointing and clicking, where the clicks causes messages to be sent). Therefore, when a plot in a spreadplot experiences some change (for example, when the user selects a variable in a window which is displaying a list of variables), it sends a message to its spreadplot message manager about the details of the change. The spreadplot message manager then forwards the message to the appropriate objects (or to all objects, if it is the first time the message has been sent). Each object, having been programmed by the spreadplot developer, knows how to respond to any message it might receive, responds appropriately.

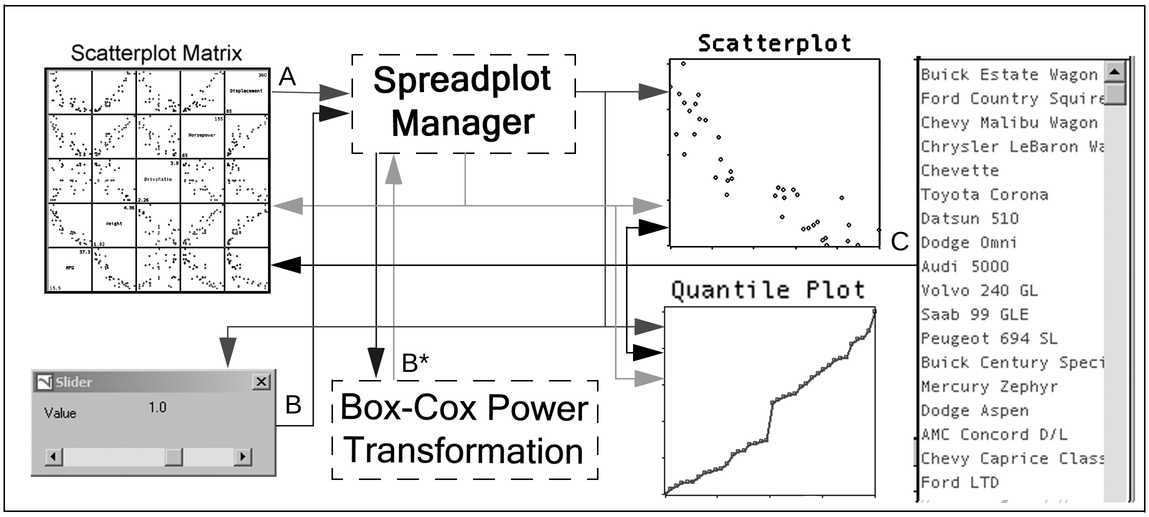

Figure 2: Schematic of Communication Architecture for Spreadplot shown in Figure 1

Now, it may be that the message sent by a plot needs to be processed by a statistical object before it is received by the group of objects. That is, the message needs to be processed so that the processed message can be distributed to the group of objects. From the developer's view point, this case is no different than the case where the message itself is distributed to the plots. If processing by a statistical object is needed, the developer must write a method for the statistical object, just as for graphical objects. Then, once the message is processed by the statistical object the results of the processing will be sent by the statistical object to the manager for distribution to the group of objects.

4.1.1 Overview of Gossip Architecture

An informal way of looking at the workings of the spreadplot message manager is to regard it as a gossipper. The manager listens to each of its objects. When it hears something, it doesn't stop to think, it just tells everyone else about it. It does pay attention, however, when one of its constituents says it doesn't want to hear about that anymore. Other than that, it doesn't do anything except pass the message on to all those that don't say they don't want to listen. Thus, the overall architecture is called the "gossip" architecture

The communication network has the structure of a wheel consisting of a hub and several spokes. The central hub is the spreadplot message manager. The plots and statistical objects are nodes at the end of the spokes attached to the hub. All objects (i.e., the plot objects and the statistical objects) send and receive messages only to and from the spreadplot manager. The objects do not communicate with each other directly, only indirectly through the manager.

In Figure 2 we show a schematic of our gossip architecture as it applies to the example discussed in Figure 1. This schematic shows that when the user clicks on the scatterplot matrix, message "A" (represented by red lines) is sent to the manager, which sends it out to the scatterplot, quantile plot and slider. This is the message to switch focal variables. Correspondingly, message "B" (blue lines) is sent by the slider to the manager (this is the message containing the parameter of the Box-Cox transformation). The manager in turn relays it to the transformation object. The transformation object processes this message and sends out reply message "B*", represented by the green arrows, to the manager which relays it to the appropriate plots. In addition, the list of observation labels (the automobile names) are linked via the standard data-linking paradigm (not via gossip-linking). Clicking on a name modifies other plots as shown.

4.1.2 The messages

Our gossip messaging system works in the following way: Assume we have created a spreadplot object named “splot”. In Lisp-Stat you send a message to this object by using the function *send*. The message can either tell the object something, or ask it something. For example, the following message tells the spreadplot that it should be 800 pixels wide by 600 high:

```
(send splot :size 800 600)
```

On the other hand, the message

```
(send splot :size)
```

asks the spreadplot how big it is. The spreadplot above would return a value of (800 600).

Our gossip architecture requires just one message. A second message is used to improve efficiency. The required message is:

```
(send object :do-spreadplot-action ‘:method-selector args)
```

This message can be sent or received by any of the objects involved in the spreadplot (i.e., the spreadplot manager, the plots and the statistical objects. The plots and statistical objects should only be able to send the message to the manager. The manager can send it to any other object.

There are two ways in which the **:**`do-spreadplot-action` message is used:

1. An **object** sends the `:do-spreadplot-action` message to the **manager**
   When the user interacts with a plot, :`do-spreadplot-action` messages are generated and sent to the spreadplot manager. Also, the :`do-spreadplot-action` message may be sent by any object, graphical or non-graphical, as a consequence of receiving messages from the manager.
2. The **manager** sends the :`do-spreadplot-action` message to the **objects.**
   When the :`do-spreadplot-action` method-selector and its arguments are sent to the spreadplot manager it forwards the method-selector and arguments to all objects, graphical and non-graphical.

To increase flexibility, the first time an object sends a message to its manager, an additional piece of information is included in the message, information that identifies the sender to the manager, and which the manager can use when it needs to send information to the object. For example, if the message:

```
(send manager :do-spreadplot-action ‘:method-selector args :hello self)
```

is sent the first time that “self” communicates with its manager, then the manager does not need to know ahead of time which objects it is managing, but learns this as time goes along.

To improve efficiency, when an object receives a message that attempts to select a method that the object does not have, the object sends a message back to the manager indicating that the message was irrelevant to the object. The manager updates its list of method-selectors accordingly, so that it will not send out the irrelevant message to the object again.

The notions above require every object to have the following `:do-spreadplot-action` method:

```
(defmeth object :do-spreadplot-action (method-selector method-arguments)
   (let ((i-have (send self :has-method method-selector))
         (splot (send self :spreadplot)))
      (if i-have
         (send self ‘:method-selector method-arguments)
         (when reply (send splot :no-method ‘method-selector self))))
```

All objects can be given this method by defining it for an ancestor near the root of the object inheritance hierarchy. This method determines whether the object has the requested method. If it does, the do-action method invokes the requested method, using the arguments that have been received. In not, when asked, the object replies to the manager that it does not have the requested method. Thus, the secondary message is:

```
(send manager :no-method ‘method-selector self)
```

Upon receiving this message, the manager updates its information so that it only sends `:do-spreadplot-action` messages with this selector to those objects which have the method needed to process the arguments. Note that this message needs to be send only once by an object, and only by objects which lack the method.

In terms of gossiping, the manager doesn’t care who it gossips with, it will take new members into the group, no questions asked, and go right ahead and repeat the gossip regardless. But, while the manager is indiscriminate about adding new members to its gossip-group, it is very concerned about not sending unwanted gossip to the members. And, the manager never wastes any time repeating itself, nor in talking to those that don’t want to hear.

4.1.3 An example

Lets return to the Power Transformation spreadplot shown in Figure 1 and schematized in Figure 2. This spreadplot has seven objects. The graphical objects include the scatterplot-matrix, quantile plot, and scatterplot, whereas the nongraphical objects include the namelist and slider, which are visible, and the transformation statistical object and the manager, which are not. Three of these objects send messages to the manager (the scatterplot matrix, the slider and the transformation object), while five of them react to messages they receive (the scatterplot matrix, the quantile plot, the scatterplot, the slider and the transformation object).

There are three different messages involved: The scatterplot-matrix sends a message which changes which variable is the focus of the spreadplot, to which the quantile plot and scatterplot react by showing the appropriate plot for the new variable, and to which the slider must adjust itself to show the transformation parameter for the selected plot. On the other hand, the slider sends a message which specifies a new value for the transformation’s parameter. This message is processed by the transformation object with the results being shown as changes in the scatterplot-matrix, quantile plot and scatterplot. In addition, though we have not mentioned it, the namelist sends messages that do not involve the spreadplot mechanism, but which use the XLispStat data-linking protocol.

Now we consider the gossip in detail. When the user clicks on, say, the lower left-hand cell of the scatterplot-matrix, the click causes the following message to be sent by the scatterplot matrix to the spreadplot-manager:

```
(send manager :do-spreadplot-action ‘:change-focal-variable “MPG”)
```

The spreadplot manager in turn relays the message

```
(send objects :do-spreadplot-action ‘:change-focal-variable “MPG”)
```

to each of the objects in the spreadplot. When an object receives the `:do-spreadplot-action` message referring to a method it does not have, the object tells the manager not to send such messages in the future. This is done by the message:

```
(send manager :no-method ‘:change-focal-variable self)
```

where `:change-focal-variable` is the method-selector in question, and `self` identifies which object is responding. The spreadplot manager maintains a set of lists which keep track of these responses so that it can prune the messaging appropriately.

Now we turn to the message originated by the slider. This message is used to change the shape of the transformation. When the user manipulates the slider, it sends a message such as:

```
(send manager :do-spreadplot-action ‘:change-transformation .35)
```

where the value .35 is the new value of the transformation parameter. The manager then relays:

```
(send objects :do-spreadplot-action ‘:change-transformation .35)
```

At this point, if this is the first time this method-selector has been sent, all of the objects except the transformation tell the manager they don’t have the requisite method:

```
(send manager :no-method ‘:change-transformation self)
```

The transformation does its computations, sending the results back to the manager via the message:

```
(send manager :do-spreadplot-action ‘:new-transformation results)
```

The manager in turn relays the message:

```
(send objects :do-spreadplot-action ‘:new-transformation results)
```

to all the objects. Finally, if this is the first time this method-selector has been sent, those objects whch do not have the required method send the manager the message

```
(send manager :no-method ‘:new-transformation self)
```

The scatterplot-matrix, normal probability plot and scatterplot must each have a `:new-transformation` method that applies the transformation in the appropriate way. In addition, the slider, normal-probability plot and scatterplot must have a `:new-focal-variable` method to respond to this specific action request when it is received. Also, the transformation statistical object must be able to respond to one of the messages. Thus, some of the objects must react to just one incoming message, while others must react to two different messages. Furthermore, the namelist object does not have to respond to either message. Despite these differences, the action method given above will work as desired. Of course, the spreadplot developer must write the methods for the relevant method selectors to select.

4.2 Spreadplot Architecture: Layout

Whereas the communication structure of a spreadplot is a wheel, the layout structure of a is a rectangular grid. At the simplest, plotcells are squares arranged in rows and columns. Such an example is shown in Figure 3 where we see a spreadplot designed to demonstrate the central limit theorem. This spreadplot consists of a plot in the upper left corner showing the population distribution; a plot showing a particular sample in the lower left; and in the two plots on the right, empirical approximations, based on several samples from the population, of the sampling distributions for the mean and standard deviation.

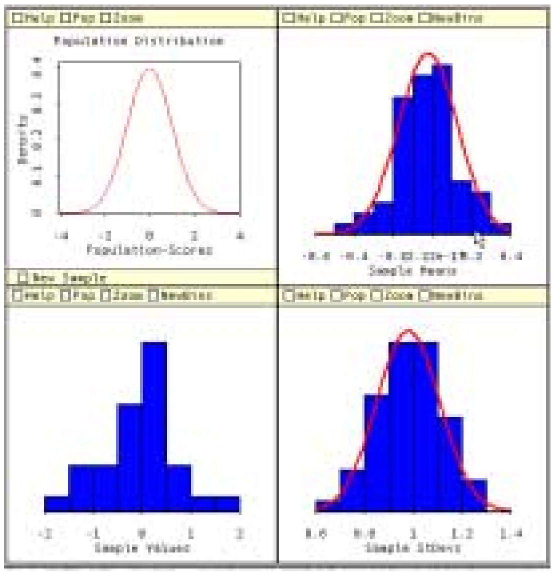

Figure 3 Spreadplot for the central limit theorem.

This spreadplot could be produced by the following pseudo-code command:

```
layout (2,2) (pop means samp stdv);
```

This command produces a spreadplot which has four plotcells arranged in a 2x2 grid of two rows and two columns. By default, all of the plotcells are square and the same size, there is one plot per plotcell, and every plotcell has a plot in it. All of these restrictions can be relaxed by using the scheme shown in table 1. With this scheme we can create the complex layout of the spreadplot shown in Figure 1 as well as the simple layout shown in Figure 2.

Consider the layout of the spreadplot in Figure 1. Looking at the figure, we see a large scatterplot matrix on the left. This large scatterplot matrix is bordered on the right by a vertically arranged pair of plots, which are themselves bordered on their right by a tall but narrow list of labels. While it may not be obvious, this spreadplot is based on a 2x4 grid whose basic plotcell size corresponds with the sizes of each plot in the vertical pair of same-sized plots. The scatterplot matrix occupies four plotcells on the left, and the labels take up two plotcells which are 1/2 the width of other columns. This layout could be created[2] by the following pseudo-code:

```
layout          2,4;
objects         (scatmat, *, npplot, labels, scatter),
                (*, *, scatter, *);
span-right      2, 0, 1, 1;
rel-widths      1, 1, 1, 1/2;
rel-heights     1,1;
supplemental    slider;
```

The layout statement specifies a grid of 2 rows and 4 columns. The objects section of the pseudocode specifies that the first row contains the scatterplot matrix, an empty plotcell (denoted *), the normal probability plot and the labels. We also see that the second row consists of two empty cells, the scatterplot, and another empty cell. The empty cells to the right and below the scatterplot matrix provide space for it to expand, while the empty cell below the labels provides space for the long labels list.

The expansion is accomplished using the information following the `:span-right` and `:span-down` keywords. To understand this, you must know that the information following the `:objects`, `:span-right` and `:span-down` keywords corresponds positionally. That is, the first `:span-right` and `:span-down` values (which are both 2) concern the first object (scatmat). These values specify how many plotcells are covered by each plot. Therefore, the scatterplot matrix occupies two rows and two columns. We can also see that the labels window spans down over the plotcell below it; that the normal-probability and scatterplot have span values of 1, meaning they occupy only their own plotcell; and that the empty plotcells occupy no space (span values of zero). Note that there can be several objects per cell. When there is, the objects in the same cell are laid out on top of each other, the first one being shown initially. The others will be brought to view via user interactions.

Relative widths and heights of plots is determined by the information following the keywords `:rel-widths` and `:rel-heights`. Cells are all the same size unless these keywords are used. When they are used, the keywords are followed by a value for each row or column. Column widths are proportional to the `:rel-widths` values, row-heights to the `:rel-heights` values. Thus, the labels window is 1/2 the width of the other columns. Finally, the `:supplemental` keyword permits adding supplemental objects to the spreadplot. Usually, these objects are sliders which enable the user to interactively input continuously variable parameter values, or modeless dialogs that are used as control panels.

2. The spreadplot shown in Figure 1 is created by the following Lisp code:

```
(spreadplot  (matrix ‘(2 4) (list scatmat      nil      npplot      labels
                                  nil          nil      scatter     nil))
             :rel-widths      ‘(1 1 1 .5)
             :span-right      ‘(2 0 1 1    0 0 1 0)
             :span-down       ‘(2 0 1 2    0 0 1 0)
             :supplemental    slider)
```

5 Discussion

A spreadplot is a visualization which simultaneously shows several different views of a dataset or model. The individual views can be dynamic, can support high-interaction direct manipulation, and can be algebraically linked with each other, possibly via an underlying statistical model.

In this paper we have presented our “gossip” architecture for spreadplots, a software architecture which simplifies the spreadplot developer’s task. The basic architectural concept is that a spreadplot consists of a group of objects, one of which is the managing object. The other objects include the graphs that the user sees, and additional objects that facilitate the interactions that the user can have with the graphs. These objects may be visible, such as a slider or a control panel, or the may not be visible, such as a statistical model or a dataset.

The gossip architecture addresses the two main problems faceing the spreadplot developer: The creation of the layout of the plots and the creation of the communication links between them. We believe our architecture provides simple mechanisms for dealing with these problems.

Of these two problems, the layout problem is the simpler, and is the one which has been worked on by previous developers (Murrel, 1999). While our approach was developed prior to the work discussed by Murrel, it is basically very similar, addressing the same problems and, for the most part, providing the same solutions.

The very essence of a spreadplot, however, lies in their between-plot communication links, not just in their arrangement on the screen. Afterall, a spreadplot is a multi-view visualization in which the views are linked together, thus there must be a communication mechanism to provide the links. In addition, a spreadplot is dynamic and supports high-interaction direct manipulation. Furthermore, the implications of the changes made by the user in one view must be instantaneously portrayed in every other view, keeping in mind that these “implications” may result from computations performed by other objects.

The fundamental characteristics of our gossip architecture are that messaging is centrally managed and pruned. Messages are always sent by an individual object to the manager. The first time that a specific type of message is received, the manager broadcasts it to all of its objects, asking each to reply about whether it has a method to process the message. In this way the manager prune objects that do not need to see the message, so that the “broadcast” becomes a “pointcast” in the future.

When compared to an non-centralized and unprunned message passing scheme, our message flow control mechanism results in fewer messages being sent. This in turn provides for improved message traffic flow and for greater control over the timing of messages. The number of messages is kept to a minimum thereby maximizing the efficiency of the communication process. The mechanism also results in simplified program structure, since no object needs to know anything about the nature of the other objects in the group, including, in particular, which ones can process the message.

In addition to being centralized and pruned, the gossip architecture is indirect, symbolic, and untyped: “Indirect” because objects never communicate directly with other objects, only indirectly through the manager; “symbolic” because communication involves a unique symbolic representation (the “method selector”) which identifies the message and selects an object’s method for the desired action; and “untyped” because the objects in the group may be any type of object. In particular objects may or may not be graphical. Those that are graphical are the plots themselves. The manager is a non-graphical object. Other non-graphical objects may include statistical objects (i.e., variable, data, transformation or model objects), other spreadplot manager objects, menu items, dialog items, etc. In short, the group of objects involved in a spreadplot can include any kind of object which is relevant to the functioning of the spreadplot. Our system is also hierarchical: Every object has a manager. Each object can only send messages “up” to its manager. The manager is itself managed, so that it can send messages “up” to its' manager, etc. Of course, managers in their turn send messages “down” to the objects they manage. Finally, our architecture is “anonymous”: The only thing that an object in the system needs to know is the identity of its manager.

Thus, in our approach, any given object, whether it be a graphical object, a statistical object or another kind of object altogether, never needs to know anything about the other objects. Indeed, it doesn’t even have to know that there are other objects. It simply needs to know who its manager is, so that it knows where to send messages, and it needs to know how to respond to messages it receives (including not to respond). That is, from the senders viewpoint, the receiving objects are totally anonymous. Of course this is true for all objects, other than the manager: No object knows anything about the objects that the message is being sent to. The converse of this is also true: No object knows anything about the objects that it receives messages from. On the other hand, all objects know everything about what is being communicated. All objects received all messages, at least until they tell the manager they don’t need to have the message sent to them any more.

Interestingly enough, the exact opposite is true for the manager: The manager knows all there is to know about all the objects in the spreadplot, but knows nothing about the content of any of the messages. It simply relays the message from the sender to the receivers, having no reason to evaluate the message itself. Note also, that the manager does not need to know what objects are involved in the spreadplot. It can simply wait until an object sends it a message, and, if an object must identify itself the first time it communicates with its manager, then the manager can learn what objects it manages.

Our spreadplot architecture has at several very important advantages: It is efficient, tractable, flexible expandable and maintainable. We discuss these next

Efficient: The gossip’s centralized communication architecture is efficient, although it may not seem so at first: After all, doesn’t every message get sent twice? Yes, but after pruning has taken place each message is sent $1+n_r$ times, where 1 is for sending the message to the manager and n_r is for the number of receivers of the message. This is only 1 more message than would otherwise be sent out. More importantly, note that the number of messages sent grows only linearly as the size of the problem increases.

Tractable: Our gossip architecture is tractable since it is always possible to identify where every message is going to, and, with 1 additional piece of information, where it is coming from. An inspector could reveal the manager’s communication matrix, a binary indicator matrix that indicates which messages are needed by which receivers. Thus, the manager can always be inspected to see what is the communication pattern at any given time. Furthermore, if the sender adds identifying information to a message, it will always be possible to inspect a receiving object to understand where it is getting messages from and what the message are. When these characteristics are combined with the linear growth rate of the architecture discussed in the previous paragraph, one understands the tractability of the architecture relative to other possibilities.

Flexible, expandable and maintainable: Our gossip architecture is flexible and it is expandable. Since no object other than the manager needs to know anything about the other objects, and since the content of messages can be anything at all, it is possible at any given point to add or remove objects and to change the messages without undue difficulty. Thus, as a system inevitably grows over time, it is straightforward to grow the spreadplot subsystems. It follows that our architecture is easily maintainable. Again, since systems inevitably grow over time, they need to be maintained in the face of the changing requirements faced by the system and in the face of the changing environments in which the system is developed. Due to the flexibility and expendability of the system, it is also highly maintainable.

It may be useful to discuss alternative architectures which could be used for managing spreadplots so the advantages of the gossip can be better understood. We can think of two alternatives for the architectures: the direct (simple) approach and a architecture based on the statistical object represented by an spreadplot.

The direct architecture would consist of plots sending messages directly to other plots. This architecture has the apparent advantage that it is not necessary to have a manager, and was used in our earliest implementations. However there is the obvious disadvantage that removing a plot breaks the messaging system. This happens because the rest of the plots now will send messages to a plot that is no longer there. Therefore, the code for every plot in the set has to be modified. This does not happen with the architecture presented in this paper because in our current architecture each plot does not know anything about the other plots in the spreadplot. Consequently, removing a plot does not have consequences on other plots.

Another possibility is to use an architecture in which a statistical object manages the spreadplot. The statistical object could know which plots depend on them, and could manage the messaging itself. This architecture has the advantage that the plots can speak directly to the statistical object without needing an intermediate object, but, on the other hand, defies the object-oriented programming philosophy of restricting objects to the management of specific domains. That is, object oriented programming teaches us that a statistical object must focus on statistical matters, avoiding as much as possible dealing with issues that fall outside of this focus. On the contrary, the architecture we suggest guarantees this separation because non-statistical issues are dealt with by the spreadplot manager, while the statistical issues are simply re-directed to the object which specializes in statistical issues.

6 Conclusion

About a decade ago, Young, Faldowsky and McFarlane (1993) introduced the concept of a spreadplot, and provided working examples of spreadplots for multivariate data, high-dimensional tourplots, multidimensional scaling and principal components analysis. Since then, there has been a significant growth in applications of spreadplots in ViSta, covering a notable number of statistical needs.

At the time this is being written, the currently released version of ViSta[3] integrates about 30 different kind of spreadplots. These include at least 11 spreadplots for exploring raw data, there being a spreadplot specifically constructed for each of several kinds of data, including: Univariate, Bivariate, Multivariate (numeric and Guided Tour), Category, Classification, Frequency Classification (one-way and n-way), Frequency Table, Crosstabulation and Data Simulation. We can also count four more in develepement. ViSta also has spreadplots for data transformations such as the Box-Cox (Figure 1), Folded Power (Young & Valero 2000) and Missing Data Imputation (Valero & Young 2000). Finally, there are spreadplots for visualizing statistical models, including Analysis of Variance, Correspondence Analysis (Bee-Leng Lee 1996) Multidimensional Scaling, Multivariate Regression, Principal Components (Young & Valero 1999), Regression Analysis (Bann 1996b), Univariate Analysis, Cluster Analysis, Frequency Analysis and Loglinear Analysis. Some of these spreadplots have been developed by those who have had no contact with others having any experience in this particularly tricky type of software development, and who have had no access, alas, to documentation. The growth in spreadplots, despite the obstacles, suggests the concept is viable.

I addition, a spreadplot editor is under development. This editor already provides access to the relatively uninitiated to all of the layout features discussed above, and it is expected that the communication features will be working long before this paper appears in print. This editor is a partially point-and-

3. ViSta can be downloaded for free from **www.visualstats.org**

click, partially code-based editor. It allows the specification of layout and communication by point-and-click actions, and only requires coding of the details of the action methods.

All this experience has shown both the rewards and the difficulties of developing multiple view systems where the highly dynamic views support instantaneous interaction with the user and instant communication between the several views. The problems in the research and development of our system over the last 10 years have lead us to try various software structures. In fact, communication architecture described herein is the fifth or sixth attempt at finding a fully viable solution to the problem. The solutions to the problems of layout and communication described in this paper are now quite stable and mature, suggesting that they are reasonably appropriate solutions. This architecture makes it simple to include/exclude new views from the set, to re-arrange the views easily, and to define and modify the interactions between the views quickly. In addition, the visualizations developed using our designs are smooth and efficient, and are ready for primetime as is, obviating the need for a prototype development stage. If, as is expected, the spreadplot editor is available when this paper is published, we foresee the rapid increase in the development and deployment of spreadplots for statistical data visualization.

7 References

Bann, C. M. (1996 a), “Statistical Visualization Techniques for Monotonic Robust Multiple Regression”. MA Thesis, Psychometrics Laboratory, University of North Carolina, Chapel Hill, NV.

Bann, C. M. (1996 b), “ViSta Regress: Univariate Regression with ViSta, the visual Statistics System”. L.L. Thurstone Psychometric Laboratory Research Memorandum.

Becker, R. A., and Chambers, J. M. (1984), *S: An Interactive Environment for Data Analysis and Graphics*, Belmont, Ca: Wadsworth.

Becker, R. A., Cleveland, W. S. and Wills, A. R., (1988). *The New S Language: A Programming Environment for Data Analysis and Graphics,* Pacific Grove, Ca: Wadsworth.

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

Becker, R. A., Cleveland, W. S. and Shyu, J. (1996), “The Visual Design and Control of Trellis Display”, *Journal of Computational and Statistical Graphics*, 5, 123-155.

Bee-Leng Lee (1996), “Correspondence Analysis”. L.L. Thurstone Psychometric Laboratory Research Memorandum.

Chambers, J. M., Cleveland, W. S., Kleiner, B., and Tukey, P. A. (1983), *Graphical methods for data analysis*, Pacific Grove: Wadsworth & Brooks.

Cook, D. R. & Weisberg, S. (1999), *Applied Regression Including Computing and Graphics*, New York: Wiley and Sons.

Emerson, J. D. (1991), “Introduction to transformations”, in *Fundamentals of Exploratory Analysis of Variance*, eds. D.C. Hoaglin, F. Mosteller, and J. W. Tukey, New York: Wiley & Sons, pp. 365-400.

Friendly, M. (1999), “Extending Mosaic Displays: Marginal, Conditional and Partial Views of Categorical Data”, *Journal of Computational and Graphical Statistics*, 8, 373-395.

McFarlane, M., & Young, F. W. (1994), “Graphical Sensitivity Analysis for Multidimensional Scaling”, *Journal of Computational and Graphical Statistics*, 3, 23-34.

Murrel, P. R. (1999), “Layouts: A Mechanism for Arranging Plots on a Page”, *Journal of Computational and Graphical Statistics*, 2, 121-134.

North, C. & Shneiderman, B. (2000), “Snap-Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata”. In *Proceedings of AVI 2000*, Palermo.

Stine, R. and Fox, J. (1997) (eds.), *Statistical Computing Environments for Social Research*, Thousand Oaks: Sage.

Stuetzle, W., (1987), “Plot windows”, *Journal of the American Statistical Association*, 82, 466 - 475.

Swayne, D. F., Cook, D. and Buja, A. (1998), “XGobi: Interactive Dynamic Data Visualization in the X-Windows System”, *Journal of Computational and Graphical Statistics,* 7, 113-130.

Tierney, L. (1990). *Lisp-Stat: An Object-oriented Environment for Statistical Computing and Dynamic Graphics.* New York: Wiley.

Velleman, P. F. (1992), *DataDesk Handbook*, Ithaca, NY: Data Description Inc.

Wang Baldonado, M. Q., Woodruff, A. & Kuchinsky, A. (2000), "Guidelines for using Multiple Views in Information Visualization", *Proceedings of AVI 2000*, Palermo, pp. 110-119.

Wills, G. (1999), "Linked Data Views", *Statistical Computing & Statistical Graphics Newsletter*, 10, 20-24.

Wilhelm, A. F. X. (1999), "A data model for interactive statistical graphics", Proceedings of the Section on Statistical Graphics. Baltimore. pp 61-70. AQUI FALTA AÑADIR LO DEL CONGRESO

Valero, P. M. & Young, F. W. (2000), "Missing Data Imputation". L.L. Thurstone Psychometric Laboratory Research Memorandum.

Young, F. W. (1992). "ViSta: The Visual Statistics System". UNC Psychometric Laboratory, Chapel Hill NC.

Young, F. W., and Bann, C. (1997), "ViSta: A Visual Statistics System", in R. Stine and J. Fox (Eds.), *Statistical Computing Environments for Social Research,* Thousand Oaks: Sage, pp. 207-235.

Young, F. W., Faldowsky, R. A., and McFarlane, M. M. (1993), "Multivariate Statistical Visualization", in C. R. Rao (Ed.), *Computational Statistics. Handbook of Statistics,* Amsterdam: Elsevier Science, pp. 959-998.

Young, F. W. & Valero, P. M. (1999), "Principal Component Analysis". L.L. Thurstone Psychometric Laboratory Research Memorandum.

Young, F. W. & Valero, P. M. (2000), "Transformations". L.L. Thurstone Psychometric Laboratory Research Memorandum.