DEPARTMENT OF STATISTICS & ACTUARIAL SCIENCE

TATISTICS

TECHNICAL

REPORT SERIES

STAT-88-21

Statistical Analysis Maps

R.W. Oldford and S.C. Peters

UNIVERSITY OF WATERLOO

Statistical Analysis Maps

R.W. Oldford
University of Waterloo

S. C. Peters
Apple Computer

Abstract

This paper presents software designed to aid the interactive management of a statistical analysis. A graphical interface is proposed which allows the analyst to keep track of the analysis and manage it as it is being carried out. The implementation is in an experimental statistical system, but the design principles apply more generally. Interactive command history lists and object-oriented programming suggest how new statistical environments can evolve from command line interfaces to graphical ones. The design is based on modelling a statistical analysis as a collection of acyclic directed networks. Nodes are 'statistical analysis objects' and arcs linking them show how one step in the analysis has led to another. The network modelling the current analysis is continually displayed and the analysis can be carried out by interacting with elements of the display. The analysis management tools and key attributes of this software model are discussed.

KeyWords: Statistical computing system, Integrated programming environments, History mechanisms, Graphical interface, Object-oriented programming

1.0 Introduction

The interactive nature of modern statistical systems allows statisticians to undertake ever more involved analyses. The data can be more fully explored; more models can be considered; alternative approaches can be undertaken. While this power encourages a more thorough analysis, it also produces an enormous amount of information. Unorganized, this information can be confusing. The problem is especially acute when the analysis must be understood, and perhaps continued, by someone other than its original author.

1.0 Introduction

A data structure containing the history of the analysis is essential to its management. But what form should this data structure take? What aspects of the analysis should be recorded, and how? Are simple history lists of commands enough?

In this paper, we present an analysis management system that addresses these questions. It is built on a history data structure that graphically displays the analysis as it progresses. The display is a collection of acyclic directed networks which we call an *analysis map*. The analyst can interact with it through a command language or more directly through a pointing device like a mouse. The design is directed towards encouraging the latter style of analysis (Oldford and Peters, 1988, describe an example session).

The proposed history structure evolves quite naturally from simpler history lists that capture the commands of the analysis system. The interactive statistical system called S (Becker and Chambers, 1984), the LISP programing language, and object-oriented programming are used to illustrate this evolution.

Section 2 provides some necessary background information. The evolution of history lists is briefly outlined and the information they contain described. Traditionally, much of this information is based on the time order of the command events. However, to use only the time order of events would be to miss important inferential information relating the elements of a statistical investigation. In particular, distinct and temporally independent analyses often shed light on a statistical investigation only when considered together. By recording only the time order of the analyses the logical connections of separate analyses is obscured.

To track the inferential connections between these events more faithfully, each is represented as a data structure which can contain pointers to other data structures. Object-oriented programming provides a simple means to introduce and to manage such logical connections and is therefore also briefly outlined in Section 2. The connections that are of interest between statistical data structures are described in Section 3.

A software structure called an `AnalysisMap` is introduced and described in some detail in Section 4. The `AnalysisMap` can be regarded as a highly interactive history

mechanism which displays the statistical objects of an analysis and the logical connections between them as a directed graph. It is intended to be an integral part of the analysis environment, which the analyst uses directly to carry out and manage the analysis.

In Section 5, those elements of the `AnalysisMap` that can be applied to applications other than statistical analysis are identified and a more abstract software structure introduced that will interactively display any collection of connected objects. Then new kinds of displays, which show networks defined by other important connections between statistical objects, are easily created as specializations of the more general software structure (using the inheritance facility available in object-oriented programming). Some of these specializations are also described in Section 5.

In the final section, we close remarking on some of the immediate advantages of this graphical representation of a statistical data analysis and on its potential application in future analysis systems.

2.0 Background

In this section we provide some background history on the use of history mechanisms in computing and introduce object-oriented programming. Two principal conclusions are drawn. First, interactive statistcal systems like S (Becker and Chambers 1984) already make a great deal of use of simple linear history lists and rather simple data structures. And second, by adopting an object-oriented approach to the design of an interactive statistical system, a much richer analysis history can be recorded and, moreover, more easily conducted and managed. The reader familiar with these areas might skip this section.

2.1 History lists

We begin by discussing the simplest history structure: a procedure which automatically records every command the analyst issues in the session. A well-known example is the S *diary* function (Becker and Chambers 1984). When invoked, *diary* causes all S commands which follow it in the session to be recorded in a system file. Figure 1 gives an example of a short session's diary.

```
1.     BodyWts <- C(6654.0, 2547.0, ..., 0.005)
2.     BrainWts <- C(5712.0, 4603.0, ..., 0.14))
3.     plot(BodyWts, BrainWts)
4.     MyReg <- reg(BodyWts, BrainWts))
5.     Coefs <- MyReg$coef
6.     abline(Coefs)
7.     plot(BodyWts, MyReg$resid)
8.     LnBodyWts <- log(BodyWts)
9.     LnBrainWts <- log(BrainWts)
10.    plot(LnBodyWts, LnBrainWts)
```

Figure 1. An S session diary

Here, two data vectors are created and assigned names in steps 1 and 2. Step 3 produces a scatterplot. Steps 4 to 6 fit a straight line to the data, save the coefficients, add the fitted line to the previous plot, and plot the residuals versus the independent variable. Steps 8 to 10 transform the data and plot the results.

The S diary records the commands in the order they were issued and nothing more. It does not, for example, record the output of the commands. While the history produced by the *diary* command can be consulted, edited at a later date, and rerun, the diary itself is not interactive. That is, it is unavailable to the analyst from within the analysis session it is recording. (Only temporary escapes from the analysis system, using ! in S, permit examination of the diary file during the analysis session.)

Becker and Chambers (1986) recently implemented a new version of the S diary called an *audit* file. In addition to recording the commands, the creation date is recorded for every data item when it is either assigned or read. In Figure 1, `BodyWts`' creation date "`27-Jan-88 11:01:03`" in step 1 would be stored in the audit file as part of the information at step 1. Similarly, in step 3 the creation date is taken from the data structure `BodyWts` and recorded on the audit file. These two dates should be the same. In this way, the integrity of the analysis can be monitored.

Moreover, as the audit file grows, it is continually processed by an audit program to produce an audit structure that can then be queried interactively by the user. One such request ask for all commands that used `BodyWts` - yielding a script of steps 1,3,4,

and 8. Queries that can be addressed by the audit structure involve when and/or where data structures (`BodyWts`, `MyReg`, ...) were created and referenced. In a multiprocess (and preferably multi-window) environment, the analysis session and the audit program could be running simultaneously as two separate processes. As the analysis continued, the audit file would be updated and the audit program would process the addition. The analyst could then switch from the analysis session to the audit process whenever she wanted to query the analysis history.

This audit approach is more active than the diary approach. With the right computing environment, it could be more interactive still. Parts of the analysis could be taken from the audit, edited, and run in the analysis session, effectively making the audit process an active tool in the ongoing analysis.

More interactive history mechanisms have been proposed recently in the statistical literature (e.g. Thisted 1986), but have long been in use in the computer science community. The earliest would appear to be the *Programmer's Assistant* (Teitelman, 1972). (A more familiar, but less powerful, interactive history list is that available in the Unix operating system (e.g. using the c-shell, csh, in BSD Unix).) Written in Interlisp (any other Lisp dialect would serve equally well), the programmer's assistant provides interactive assistance to the programmer by allowing immediate interactive access to the session history from within the session.

Like S, Interlisp is interactive - commands are interpreted and executed as soon as they are complete. Figure 2 shows the script of an Interlisp session which performs the same analysis as S does in Figure 1. Each command is a list of tokens within parentheses - the first element is the function, or command, name and the remaining elements are its arguments. Each list is called a form and, as Figure 2 illustrates, forms can be nested with innermost forms evaluated first. Assignment is achieved by the `SETQ` function, so that statement 1 assigns the result of the `C` function to `BodyWts`. The `@` function in step 5 plays the same role as `$` does in S - it selects a named component (`coef`) from a data structure (`MyReg`).

```
1.    (SETQ BodyWts (C 6654.0 2547.0 ... 0.005))
2.    (SETQ BrainWts (C 5712.0 4603.0 ... 0.14))
3.    (SETQ MyPlot (PLOT BodyWts BrainWts))
4.    (SETQ MyReg (REG BodyWts BrainWts))
5.    (SETQ Coefs (@ MyReg coef))
6.    (ADDLINE MyPlot Coefs)
7.    (SETQ Plot2 (PLOT BodyWts (@ MyReg resid)))
8.    (SETQ LnBodyWts (LOG BodyWts))
9.    (SETQ LnBrainWts (LOG BrainWts))
10.   (SETQ Plot3 (PLOT LnBodyWts LnBrainWts))
```

Figure 2. Script of LISP commands

All LISP functions return a value. Even the `PLOT` function of line 3 returns a value that is then assigned to `MyPlot`. The value returned might be a number, an array, a tree structure, or even another function. In LISP, there is essentially no restriction.

The programmer's assistant records, on a history list, each command entered *and* its value. The programmer's assistant will also respond to its own set of commands which allow user interaction with the history list. A programmer's assistant command is invoked like any other, by typing it directly to the system executive. Effectively, the programmer's assistant acts as an intermediary, albeit usually invisible, between the user and the LISP executive that evaluates LISP expressions. The programmer's assistant deals with its own commands directly and calls the LISP executive only as necessary.

Some simple examples of programmer's assistant commands are he following. `REDO 5` will cause statement 5 to be executed again (becoming statement 11 in the history). `UNDO 5` will undo the effect of statement 5 (detaching the value assigned to `Coefs`). Typing '`USE LnBodyWts FOR BodyWts AND LnBrainWts FOR BrainWts IN 4 TO 7`' will repeat statements 4 to 7 (becoming new statements 11 to 15) after substituting `LnBodyWts` for `BodyWts` and `LnBrainWts` for `BrainWts`. And, `FIX 5` will display a copy of statement 5 to be edited and evaluated.

The set of programmer's assistant commands is large and extendable. Moreover, the programmer's assistant can itself be accessed from other programs. For a complete description of its power, see Teitelman (1972, 1977), and Xerox (1985).

The programmer's assistant is the least passive of the history mechanisms described here. It achieves its high level of interaction by being an integral part of the analysis environment. As its name suggests, its aim is to aid the programming task. While this implies that it also aids the statistical data analyst, it also necessarily restricts its scope as an analyst's assistant. A programmer's assistant must not make use of information that may be peculiar to statistical data analysis.

We now turn to the statistical data structures that are produced in the course of an analysis. The next subsection shows how richer data structures called objects can be easily introduced into a statistical analysis software environment. Relationships between these objects will be shown to provide analysis information that can be used to assist the analyst in managing, and hence conducting, an analysis.

2.2 Data structures and object-oriented programming

The analysis history is more than a history of the commands used and their arguments. It is also a history of the data structures that were created and explored.

In the sample S session (Figure 1), a number of different data structures were generated. The initial *vector* structures `BodyWts` and `BrainWts` were created and from these the more complex data structure `MyReg` was formed. `MyReg` is a *hierarchical data structure* that contains several named component data structures, including `coef` and `resid`, which can be extracted using the `$` function (see Becker and Chambers, 1984).

Similarly, every form in the LISP session (Figure 2) returns some structure as its value. The `C` function returns a vector structure. The `REG` function returns a regression structure with components `coef` and `resid` (and possibly others). Statement 3 shows that `PLOT` returns a plot structure. (Similar structures exist in S but cannot be assigned to tokens like `MyPlot` in Figure 2.) The `ADDLINE` function in statement 6 takes the plot `MyPlot` as one of its arguments. This is very convenient in environments where many different plots can be displayed and interacted with at once (e.g. see Stuetzle 1987). The analyst may want to refer to previous plots at a later time in the analysis.

In both S and LISP, each function is only applicable to certain kinds of data structures. The log function can be applied to `BodyWts` but not to `MyReg`. Other functions are more generic, applying to more than one type of data structure. For example, `abline` in S will accept two numbers, or a vector of two numbers, or any hierarchical structure that contains a `coef` component (e.g. `MyReg`). Because it is useful to have different kinds of data structures for different kinds of statistical results (vectors, regression results, plots, etc.) and because many functions are designed to operate on certain structures and not others, it would be convenient if these data structures and the functions designed to operate on them were more closely associated.

A style of programming called object-oriented programming is designed to provide this convenience. Functions can be directly attached to a data structure type. Functions and data are bound together in a single structure called an *object*.

An object can be thought of as a hierarchical structure with named components called its *Instance Variables*, or *IVs*. Unlike an S hierarchical structure, however, there are also *methods* associated with each object. These methods are the functions commonly applied to that object. A `PLOT` object, for example, would have an *AddLine* method which, when invoked, would take the necesary slope and intercept information from the user (analyst or program). A method is invoked on a given object according to the following syntax:

```
(<- Object Method Argument1 Argument2 .. ArgumentN)
```

Figure 3. Message passing syntax

This syntax is sometimes called "message passing". The idea is that each method on `Object` can be invoked by passing `Object` the name of the method (the message) together with arguments for that method. The arrow symbol, `<-`, is read as "send" so that the whole statement can be read as "send `Object` the message *Method* with arguments `Argument1` to `ArgumentN`". The flavour here is that the data structure (`Object`) is the active agent. It receives the message *Method* and invokes the corresponding method function.

Since many objects will share basic structure, any common structure is recorded in special objects called *classes.* Individual objects are then taken as *instances* of a class. For example, `BodyWts`, `BrainWts`, `LnBodyWts` and `LnBrainWts` would all be instances of a *class* called `Vector`.

Each class specifies the instance variables, their default values, and the methods that every instance of that class must have. Classes (e.g. vector) are the templates used to construct other objects (`BodyWts`, `BrainWts`, etc.). New instances of a class are created by sending the class object the *New* message. Once created, these instances will persist in the virtual memory of the machine. Many different instances of the same class can be created. These will have the same methods and instance variables, but the values of their instance variables may be different. (See Stefik and Bobrow, 1985, for a more general treatment of object-oriented programming.)

For example, the general structure of a regression result might be specified by a class called `REG`. In `REG` it is specified that all instances will have IVs `coef` and `resid`. Further, if every instance of `REG` must be able to respond to the message `PrintTStats`, then the function which achieves this is defined as a method in the class `REG`.

Using an object-oriented approach, our analysis session would proceed as in Figure 4.

```
1.    (<- (C 6654.0 2547.0 ... 0.005) SetName BodyWts)
2.    (<- (C 5712.0 4603.0 ... 0.14) SetName BrainWts)
3.    (<- (<- PLOT New BodyWts BrainWts) SetName MyPlot)
4.    (<- (<- REG New BodyWts BrainWts) SetName MyReg)
5.    (<- (@ MyReg coef) SetName Coefs)
6.    (<- MyPlot AddLine Coefs)
7.    (<- PLOT New BodyWts (@ MyReg resid))
8.    (<- (<- BodyWts LOG) SetName LnBodyWts)
9.    (<- (<- BrainWts LOG) SetName LnBrainWts)
10.   (<- (<- PLOT New LnBodyWts LnBrainWts) SetName Plot3)
```

Figure 4. Message passing history

On the surface, this session would seem to be more complicated than the original S session. However each line here creates and manipulates objects which, as later sections

will show, permit a highly interactive graphical presentation of the analysis session to be relatively easily built (Section 4).

In detail, the above session can be described as follows. The `c` function returns an object that is an instance of the class `FloatVector` (a vector of floating point numbers). This instance is sent the *SetName* message (a message understood by all instances) with the argument `BodyWts`. Thereafter the instance can be referred to by its name, `BodyWts`. Statement 3 creates an instance of `PLOT`, with `BodyWts` and `BrainWts` as the values of the x and y coordinates, and then names the object `MyPlot`. Later, in statement 6, `MyPlot` is sent the message *AddLine* with the argument `Coefs`. The method *AddLine*, stored on the class `PLOT`, draws a line on `MyPlot` with slope and intercept taken from `Coefs` (as in the S example, different kinds of arguments could be passed to the *AddLine* method). Note that `(@ MyReg coef)` could have been used in place of `Coefs` in statement 6 - either way the same unique object is accessed. Unlike the hierarchical data structures in S, two different objects can share a great deal of structure.

The approach here emphasizes the software structures that are manipulated and created at each step, not the command lines. The structures are the active agents. Each one is unique, persistent in memory, and can be shared by many structures. Moreover, there are natural connections between them. In the next section, these connections are explored and formalized to become the basis for a highly interactive graphical interface to an analysis.

3.0 Connected statistical objects

An object-oriented approach emphasizes the history of a statistical analysis session as the history of statistical objects that are created and manipulated throughout the analysis. The time sequence of commands is no longer paramount. Rather, different relationships between individual objects can be considered. Our objective is to assist, or at least to cooperate with, the analyst in managing and conducting the analysis. To meet this objective the logical flow of the analysis must be recorded. The question is how can these statistical objects be related, one to another, so as to best capture this information?

One important relationship is a consequence of an object's definition. Every object is directly connected to any other object that is the value of one of its instance variables. This means that `MyReg` is connected to the vector object that is the value of its `coef` IV. Since this vector object is unique, naming it `Coefs` in statement 5 does not alter its relationship with `MyReg`. Moreover, we may distinguish those IVs (like `X` and `Y` of a `REG` object) whose values (`BodyWts` and `BrainWts`) are required to create the object (`MyReg`), from those whose values are attached after the object has been created (e.g. tthe value of `coef` in `MyReg`). We call the first kind of IV a `RequiredIV`. Should we wish to reproduce an analysis, the `RequiredIVs` would identify the necessary inputs.

Causal relationships also exist between objects which are not required components of one or the other and hence are not captured by `RequiredIVs`. For example, consider statements 8 and 9 of Figure 4. `LnBodyWts` is created as a direct result of the *LOG* message being sent to `BodyWts`. The relationship is causal and should be captured in the history.

These relations can be recorded in a generic way by attaching two new instance variables to all statistical objects - one called `CausalLinks` to record on the antecedent object the identity of the consequent object, and one called `BackCausalLinks` to record the identity of the antecedent object on the consequent object. The value of an object's `CausalLinks` IV is a list of those statistical objects which have been created as a direct consequence of a message received by that object. The `CausalLinks` IV of `BodyWts`, for example, is a list containing the single element `LnBodyWts`. If, at some later time in the analysis, `BodyWts` is sent the *SQRT* (square-root) message yielding `SqrtBodyWts`, say, then `BodyWts'` `CausalLinks` IV will be a list of two items - `LnBodyWts` and `SqrtBodyWts`. `BackCausalLinks` are entirely analogous. This makes it possible to begin at any statistical object and, from it, trace the causal sequence of events forward, or backward.

The establishment and updating of these links is made automatic by a simple modification to the send function, `<-`. We chose to define a new function `<-~`, read "send and link", to be used in place of `<-` everywhere in Figure 4. With `<-~`, whenever the result of the method is another statistical object, a link is established from

the object receiving the message to the object returned. (Note. Some messages, like *SetName*, do not 'return' any object and consequently make no links.)

Consider now the information remaining in a temporal record of events after recording the component relationships and causal links as above. Some of it is of no value at all - `BodyWts` and `BrainWts` are temporarally independent events in the sense that the logic of the analysis is independent of whether `BodyWts` or `BrainWts` was created first. Worse, some important information is not at all apparent from a temporal record. In our sample analysis the analyst may have decided to take logarithms of the data, in statements 8 and 9, on the basis of what was seen in the scatterplot of statement 3. Such information, while not deducible from the sequence of events, could be easily recorded by linking the three objects. However, `CausalLinks` seem inappropriate - the logged data were not produced as a direct consequence of any action taken on the scatterplot. These weaker relationships are captured on an IV called `AnalysisLinks` that is attached to every statistical object.

Analogous to causal links, each statistical object has both an `AnalysisLinks` IV and a `BackAnalysisLinks` IV. These links represent the logical flow of the analysis, *as perceived by the analyst.* The idea is that an analysis link should be established from one object to another, if the analyst feels that the analysis was directed from consideration of the first object to consideration of the second. This is almost always the case when the second object was created as a result of a message passed to the first. As a convenience, then, `AnalysisLinks` are also constructed automatically whenever `CausalLinks` are constructed, again by modifying the `<-~` function. The difference between the two types of links is that the analyst can make and break `AnalysisLinks` at will (each statistical object responds appropriately to the messages *MakeAnalysisLink* and *BreakAnalysisLink*). Thus, in the example session `AnalysisLinks` would automatically be established from `BodyWts` and `BrainWts` to `LnBodyWts` and `LnBrainWts`, respectively (as would `CausalLinks`). However, to indicate the logical flow of the analysis from the scatterplot, `MyPlot`, to `LnBodyWts` and `LnBrainWts`, it would be necessary to send `MyPlot` the *MakeAnalysisLink* message with `LnBodyWts` and `LnBrainWts` as arguments.

We propose, then, that three distinct connections between statistical objects be recorded on every object. First, there are the `RequiredIVs`, to distinguish those components of an object whose values are required at creation time. Second, there are the causal links (forward and backward) to indicate which object caused another to be created. And third, analysis links are meant to reflect the logic of the analysis - as determined by the analyst (with some help from the system). The first two, together with an object's creation date (also an IV), provide valuable auditing information. They are particularly important should we decide to reproduce an analysis on different data. The third, however, is of principal value to the analyst - they are the sole responsibility of the analyst and should be made and broken with care. Consequently, the tools developed in the next section are primarily designed to aid the management of analysis links.

4.0 AnalysisMaps

The previous section showed that a rich analysis history can be modelled as a set of multiply connected statistical objects. This has been accomplished by moving the relevant information from a separate history mechanism to the data structures themselves. Under this model, analysis management tools can be thought of as intermediaries that facilitate the exchange of information between the analyst and the objects created. One such intermediary is a software structure which we call an `AnalysisMap`.

An `AnalysisMap` allows the analyst to view, and interact with, any set of statistical objects and the `AnalysisLinks` between them. `AnalysisLinks` are used here because they are directly controlled by the analyst and hence play a primary role in managing the analysis. Intermediaries focussing on other connections are discussed in section 5.

Following the `AnalysisLinks` as arcs, the statistical objects are the nodes of a possibly disconnected directed graph. The whole analysis is a collection of such digraphs so that managing the analysis amounts to manipulating and rearranging these digraphs to best show the analyst's logic. These networks are displayed in an `AnalysisMap` as in Figure 5 below.

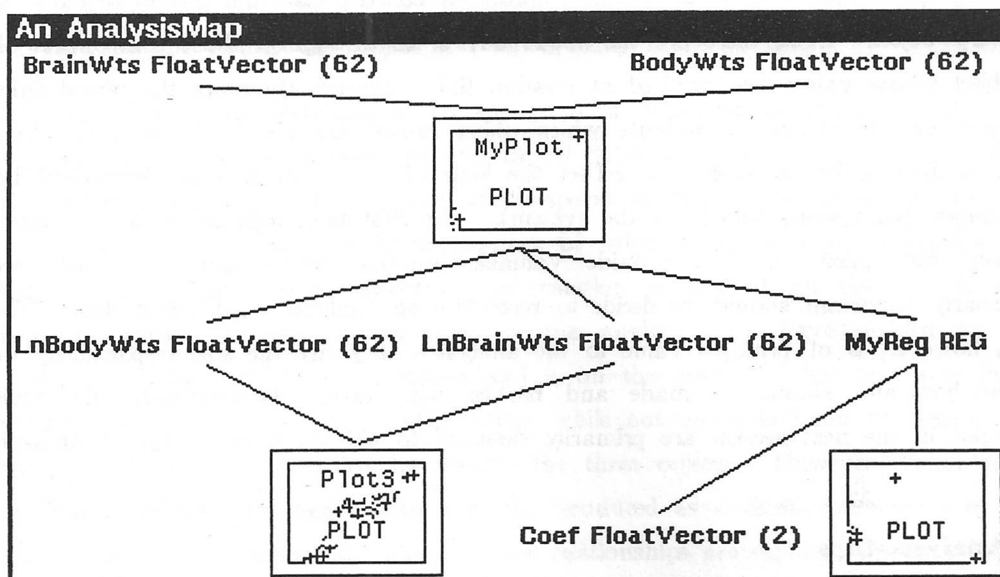

Figure 5. An `AnalysisMap` of the sample session

Every node is easily identified with an object from the session of Figure 4. The node labelled `BodyWts FloatVector (62)`, for instance, corresponds to the object named `BodyWts` in that session. The default labelling displays the class name of that object preceded by its unique name (e.g. `BodyWts`) if it has one. In general, the node labels are designed to contain as much information about the identity of underlying object as seems reasonable. In particular, the node label of each `PLOT` is a miniature reproduction of the plot that was drawn, augmented by the object's name and class.

Each object is responsible for the production of its node label. This not only makes it easy to have a label tailored to each object, but it also simplifies the design of `AnalysisMap`. For example, rather than maintain a record of the label generating functions for every object class in the definition of `AnalysisMap`, each object can be queried directly for its label. The arcs connecting the nodes are similarly determined - the `AnalysisMap` asks each object for its `AnalysisLinks`. This style of delegating responsibility is adopted wherever possible and results in a simple design for `AnalysisMap` and similar software structures (see Section 5).

Note that the `AnalysisMap` does not show all the statistical objects that were created in the analysis of Figure 4. The residuals that were accessed in line 7, for example, do not appear. Since the residual vector is an instance variable of `MyReg` it is directly available from the node `MyReg`. Unless explicitly added to the `AnalysisMap` (e.g. `Coefs`), instance variables do not appear so that the amount of detail displayed is minimized.

The uniform way of accessing such detail is through the *Zoom* message which is understood by all statistical objects. For example, to display the residual vector, and other IVs of `MyReg`, the *Zoom* message is sent to `MyReg`. This causes `MyReg` and its IV connections to be displayed as in Figure 6.

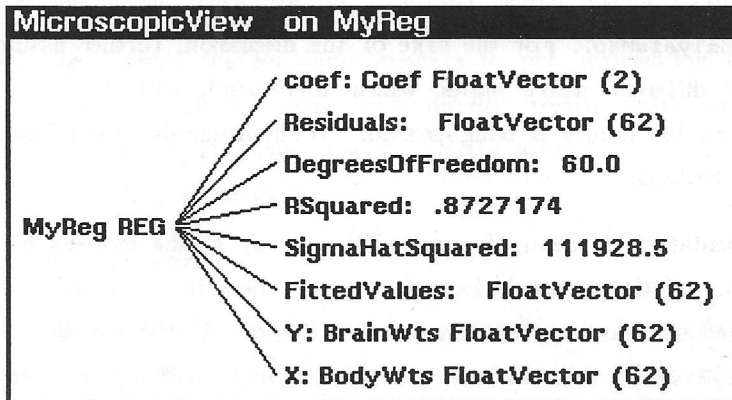

Figure 6. The `MicroscopicView` of `MyReg`

Because it is the responsibility of the object to respond to the *Zoom* message, its response can be tailored. The default response for all classes is to produce and display a software structure called a `MicroscopicView` as in Figure 6. Other objects will produce a more meaningful display - a `FloatVector` will display the numbers it contains in an array, a `PLOT` its original plot. These responses are simply implemented by specializing the *Zoom* method of the appropriate class of object (e.g. `FloatVector` and `PLOT`). If no specialization is done, the default *Zoom* method will be inherited and will produce a `MicroscopicView` as before. Consequently, the `AnalysisMap` can be

unaware that different responses exist - it simply send the appropriate object the *Zoom* message.

4.1 Interactive graphical analysis management

As described so far, an `AnalysisMap` is still a relatively passive tool for analysis management. It could be implemented as a static display on traditional hardware. However, by using a modern workstation's multiple-windowed bit-mapped display and a "mouse" pointing device, the `AnalysisMap` can be made a highly interactive tool for analysis management.

First, take the `AnalysisMap` to be a "mouse-sensitive" window, meaning simply that something can happen if the mouse button is pressed while the mouse is within the window of the `AnalysisMap`. For the sake of the discussion, further assume that the mouse has three different active states which correspond, say, to which of three different buttons on the mouse is being pressed. We'll distinguish these buttons as left, middle, and right buttons.

In our implementation, a menu of some kind pops up at the mouse's location when a button is down. If the mouse is located over an object in the map, then the menu will offer some actions that can be taken on that object. If the mouse is on the title bar of the `AnalysisMap`, then actions are offered that are applicable the displayed network as a whole. The actions, and hence contents of the menus, depend upon which of the three mouse buttons is depressed.

4.1.1 Exploring the nodes of the graph

Pointing at an object in the `AnalysisMap` and clicking the left button causes the following menu to pop up.

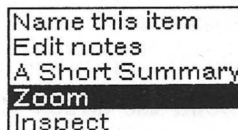

Figure 7. The object's left-button menu

The mouse is used to select a menu item by placing it over the item and releasing the mouse button. In the figure, 'Zoom' is shown selected. Once an item is selected, the `AnalysisMap` sends a message corresponding to that item to the object - selecting Zoom causes the `AnalysisMap` to send the object the *Zoom* message.

No matter which object is selected, if the left-button is used the same menu (Figure 7) is produced. This has two implications. First, the menu contents can be stored as part of the `AnalysisMap` structure. And second, every statistical object must respond to these messages. The contents of this menu are therefore chosen to be rather generic actions.

The most generic actions are those which simply exchange information between the analyst and the object. For example, 'Zoom' sends the *Zoom* message to get more detail, 'Name this item' sends the *SetName* message to put information on the object, and 'Edit notes' lets the analyst access and directly record notes on the selected statistical object (on a `Notes` IV using the *EditNotes* message). The last one is an important tool for recording an analysis.

4.1.2 Managing the network

A similar simple exhange exchange of information is possible on the whole `AnalysisMap` by pressing the left button with the mouse on the `AnalysisMap`'s title bar. A menu is produced whose items, when selected, will either produce information on how the `AnalysisMap` works, or, accept information (e.g. name for the `AnalysisMap` or notes) to be recorded on the `AnalysisMap`.

Tools for managing the relationships between objects in the analysis are found by pressing the middle mouse button while over the title bar. This produces a set of nested menus, organized by the type of actions they allow. The first level of these menus is shown in Figure 8. Arrows indicate that another menu, containing more specialized actions, will appear if the mouse is moved to the right off the highlighted item.

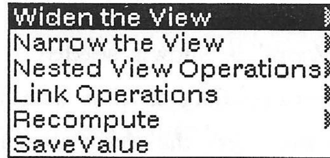

Figure 8. `AnalysisMap` title bar middle-button menu.

These actions are best understood by regarding the `AnalysisMap` as a window, or `View`, on a collection of linked statistical objects. Then, it is expected that this view could be widened, adding more statistical objects to the `AnalysisMap`, or narrowed, removing objects from the map. These actions are the first two menu items.

Many links were made to arrive at the network in Figure 5. Some were also broken (e.g. the analysis links from `BodyWts` to `LnBodyWts`). The result was a reasonably clear display of the flow of the analyst's logic (from top to bottom in Figure 5). The fourth menu item in Figure 8 provides access to those functions which allow the analyst to make and break analysis links between the displayed objects.

Moving the mouse to the right along this item produces the menu system of Figure 9.

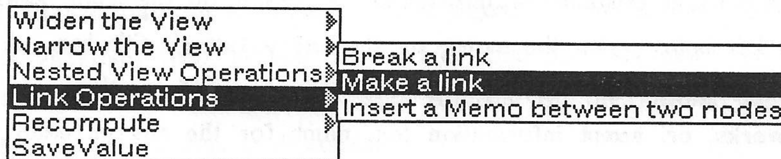

Figure 9. Analysis link operations

Selecting 'Make a link', the analyst is prompted for two statistical objects to be linked. These can be indicated by pointing to them with the mouse. This greatly facilitates the analysis management. If the analyst was required to type in the message passing expression to achieve the linking, it is not likely that much organization would go on - it would slow down the analysis too much. The remaining items allow the `AnalysisLinks` to be broken between two objects and an object called a memo to be inserted between any two nodes. `Memos` are objects that contain only notes.

4.1.3 Multiple analyses

An `AnalysisMap` is a view showing which objects are the current focus of the analysis. If so, then why not have many `AnalysisMaps`? Each could be identified with a part of the analysis that formed a separate focal point within the overall analysis. These would be separate subanalyses within the larger analysis. In turn, these subanalyses might themselves contain yet finer subanalyses. In our implementation, different `AnalysisMaps` are different instances of the `AnalysisMap` class. Each instance has an IV called the `DisplayList`, whose value is a list of the statistical objects it contains and displays. `AnalysisMaps` appearing on this list indicate subanalyses.

A statistical object might appear in the view of more than one `AnalysisMap`. For example, suppose many independent analyses are undertaken on the same data. When considering any one of these analyses, it would be useful to have this data appear in the corresponding `AnalysisMap`. This is simply achieved by having a pointer to the data object (e.g. its name) appear on the `DisplayList` of those `AnalysisMap` objects which contain the data object. The data are not copied into two different `AnalysisMaps`. If an object appears in more than one `AnalysisMap`, then it is equally accessible from each `AnalysisMap` (Zooming etc.). In a multi-window environment, many different `AnalysisMaps` can be displayed simultaneously, allowing the analyst to switch concentration from one to another as necessary.

Some analyses are best described as subanalyses - they address a subproblem of the larger analysis. In the larger problem context, the details are only of interest aggregated as the result of the subanalysis. To accomodate such aggregation graphically, `AnalysisMaps` can also contain other `AnalysisMaps`.

Moving the mouse across 'Narrow the View' produces the following menu system.

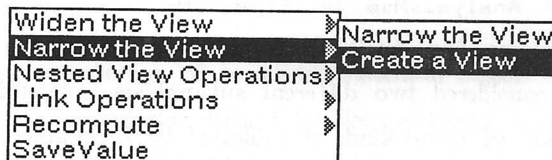

Figure 10. Creating a subanalysis

Selecting 'Create a View' produces a small icon representing an AnalysisMap inside the current AnalysisMap. The analyst is then prompted for the statistical objects which are to be included in the new, nested, AnalysisMap. The part of the network corresponding to the selected objects is collapsed and replaced by a single node represented by the AnalysisMap icon. Figure 11 shows the result for our example analysis after two subanalyses were created.

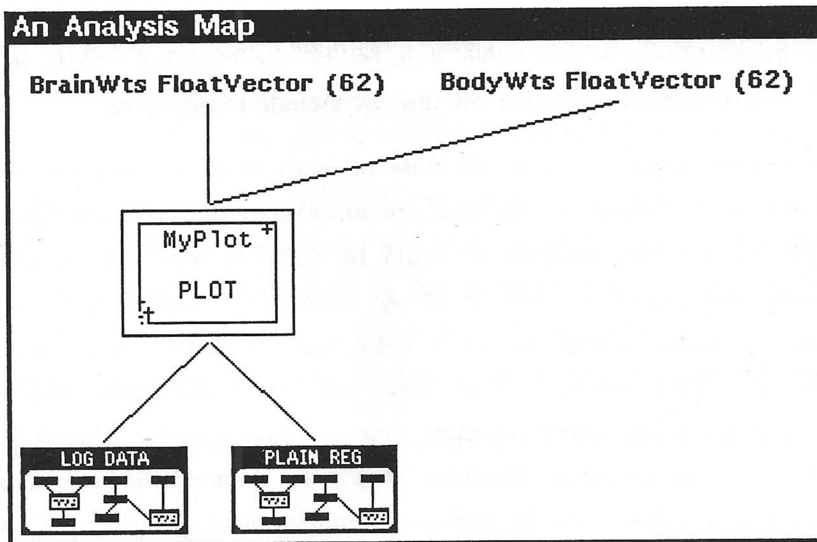

Figure 11. Analyses within analyses

These nodes behave like any other in an AnalysisMap. The left button menu is as before, allowing the subanalyses to be named, to have notes added to it, and so on. For example, the subanalyses in Figure 11 have been given names, LOG DATA and PLAIN REG, which appear in each icon's title bar.

From the top level AnalysisMap of Figure 11, the logic of the analysis is straightforward and contains few details. The analyst began with two data vectors, plotted them, and then considered two different subanalyses independently. One was a straight forward regression of some kind, as indicated by the name PLAIN REG, and the other was an analysis involving the logged data.

The detail can be returned by exploding the nested `AnalysisMap`. This is done by selecting the menu item shown in the figure below.

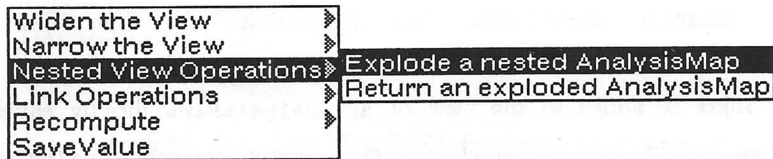

Figure 12. Operations on nested `AnalysisMaps`

As Figure 12 shows, the inverse operation should also be available.

Alternatively, the analyst can zoom in on the subanalysis by selecting 'Zoom' from the left-button menu (Figure 7). Zooming in on the `PLAIN REG` subanalysis opens up a window, as in Figure 13.

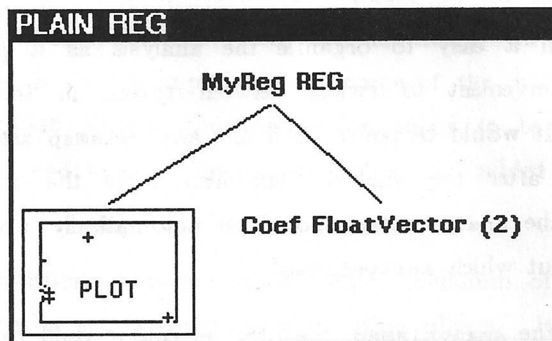

Figure 13. The subanalysis `PLAIN REG`

This too is an `AnalysisMap` object. It has the same behaviours as the larger `AnalysisMap` that contains it. The analyst carry out the subanalysis in it with none of the details appearing in the larger `AnalysisMap`.

It is as if the entire collection of connected statistical objects are displayed as a Venn diagram having the set boundaries defined by the `AnalysisMaps`. Many `AnalysisMaps` can contain the same objects, and `AnalysisMaps` can be nested within one another to

an arbitrary depth. The only difference is that the contents of a nested `AnalysisMap` are not visible inside the larger `AnalysisMap`.

This Venn diagram organization has implications for the behaviour of `AnalysisMaps`. Removing an object from one `AnalysisMap` (by narrowing its view) implies that the object be added to the view of all `AnalysisMaps` directly enveloping the first `AnalysisMap`. Like the set boundaries in a Venn diagram, `AnalysisMaps` are separate from the statistical objects they view. As a consequence, there is no reason for `AnalysisMaps` to have formal `AnalysisLinks`. Instead, arcs to and from nested `AnalysisMaps` are drawn if, and only if, a statistical object in the larger `AnalysisMap` has an analysis link to at least one object inside the nested `AnalysisMap`. Finally, care must be taken in the definition of the methods of the `AnalysisMap` class to prevent an instance from containing itself through some chain of nested `AnalysisMaps`.

4.2 Conducting the analysis

The `AnalysisMap` makes it easy to organize the analysis as it progresses. It is, however, somewhat inconvenient to switch between typing in the commands and managing the analysis. It would be preferable if the `AnalysisMap` assisted in managing the analysis, not only after the analysis, but also while the analysis was being conducted. Minimally, the `AnalysisMap` should be automatically updated whenever a command is executed. But which `AnalysisMap`?

To uniquely identify the `AnalysisMap`, the LISP executive could be modified to have a specified `AnalysisMap` updated with those objects created by the commands. This is a common approach taken with history lists (see Section 2.1). A simple command could redirect the output to different `AnalysisMaps` as appropriate.

An alternative approach is to have a selected `AnalysisMap` invoke a LISP executive as needed. In our implementation, a menu item from the title bar of any `AnalysisMap` allows this possibility. Figure 14 shows the menu item to be selected.

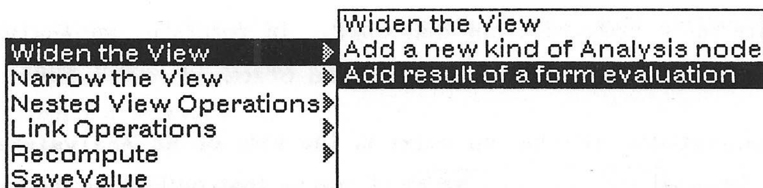

Figure 14. Invoking LISP from an `AnalysisMap`

Selecting this item causes a small window to be attached to the top of the `AnalysisMap` where lisp commands may be typed directly. The object returned by the command is added to the view of the `AnalysisMap`. This approach has the distinct advantage that the most interesting part of the analysis is immediately before the analyst when the command is entered. This encourages the analyst to a more active involvement in the analysis management.

Statistical objects can also be created and added to the `AnalysisMap` by selecting 'Add a new kind of Analysis node' from the same menu. In the `AnalysisMap`'s LISP window, the analyst is then prompted for the name of the class of object to be created. Once created, the object prompts the analyst for values to be assigned to its required variables (its `RequiredIVs`). Finally, the object is added to the view of the `AnalysisMap`.

Thus a new statistical object is created with a minimum of typing on the part of the analyst. This is a simple consequence of using objects. It also results in a more interactive `AnalysisMap`. By making the `AnalysisMap` even more interactive, and relying heavily on the fact that objects are being manipulated, the typing required to perform actions on existing objects can also be substantially reduced.

Recall that every command that operates on a statistical data structure (vectors, plots, regression structures, etc.) will operate on some structures and not on others. In Section 2.2, these commands were implemented either as methods of the objects on which they operate (like *LOG* on `FloatVectors`), or as separate objects themselves (e.g. `REG`). For the latter, the above procedure to add a new object is sufficient. For the former,

the methods are easily accessible from that object. In particular, an `AnalysisMap` can make them available by simply querying the selected object.

In our implementation, selecting an object in the view of an `AnalysisMap` with the middle button depressed will pop up a series of menus that outline the methods available for that object in an ordered fashion. The topmost level has three categories, as shown in Figure 15.

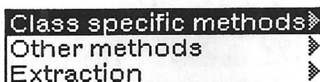

Figure 15. Top level middle-button menu for any selected object.

The first of these leads to the methods that are designed specifically for that class of object. Within this category, the methods are further organized to simplify location of the desired method. For example, the coefficient estimates and t-statistics from `MyReg` would be printed nicely by selecting the menu item as shown in Figure 16.

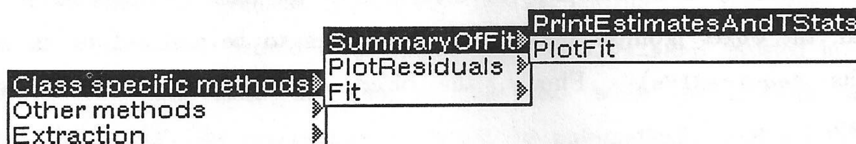

Figure 16. Middle-button menu for a `REG` object.

This would cause the `AnalysisMap` to send `MyReg` a message called *PrintEstimatesAndTStats*. Similarly, the fitted line could be plotted, various residual plots produced, and so on. If an object is returned as the value from sending the message, then it is automatically added to the view of the current `AnalysisMap`.

The second top level category leads to those methods which are less frequently used, where frequency of use is determined by us, the designers. These methods also have a second level of organization.

The third category, 'Extraction,' has nothing to do with methods. Instead, it provides a simple means to extract the values of instance variables. Figure 17 shows the menu system for `MyReg`.

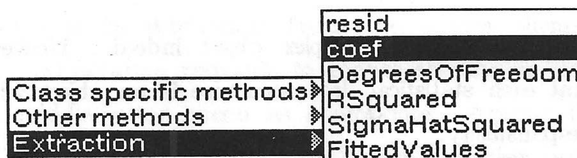

Figure 17. Extracting IVs of `MyReg`

As can be seen, the IV names are presented as menu items. Selecting one will return the value of that IV for `MyReg` and add it to the `AnalysisMap`. An analysis link will also be formed from `MyReg` to the extracted value. In our example session, this was how the vector `Coef` came to appear in the `AnalysisMap`.

Thus there are four possible ways to continue an analysis. First, new analysis objects can be created and added to an `AnalysisMap` by selecting 'Add a new Analysis node' in the title bar of the current `AnalysisMap`. Second, once a few objects exist, the analysis can continue by directly interacting with these objects in the display. If the methods are well designed, this can make the analysis progress at a rapid pace. For example, in Figure 13 residual plots could be produced by pointing directly at the `MyReg` node and selecting the appropriate method. Third, for commands that involve multiple operations, the LISP executive can be invoked above the `AnalysisMap` and the command typed in. Fourth, whenever it was desired to do many commands before adding the result to the `AnalysisMap`, the LISP executive can be used directly (in a multi-window environment, one window can be dedicated for theLISP executive).

The net result of these four possibilities, together with the analysis management tools discussed earlier, is that the analysis itself becomes something tangible. It grows in many directions and at many levels of detail. Oldford and Peters (1988) describe a sample analysis in some detail. Because the analyst nearly always interacts directly with the `AnalysisMap` to carry out the analysis, and because the analysis can be easily

and speedily carried out, the analyst is more likely to make the effort necessary to properly manage the analysis.

4.3 Responsibilities of the AnalysisMap

The AnalysisMap sounds like a very complex object indeed. However, it is greatly simplified by the fact that each statistical data structure is an object which can be made to share much of the responsibility.

If an object is selected with the left-button depressed, then the AnalysisMap produces the menu of Figure 7. When an item is selected, the AnalysisMap sends the object the corresponding message (*Zoom*, *EditNotes*, *SetName*, etc.) and its responsibility ends.

If the middle-button is depressed, then the AnalysisMap asks the selected object for the menu to produce. Once an item is selected, a message is sent to the object. The AnalysisMap observes the value returned by the object in response to the message and, if it is another object, the AnalysisMap adds it to its view. Links are automatically established by having the AnalysisMap send all messages using the <-~ function.

Even to compute the displayed network, much is handed off to the individual objects. The AnalysisMap has a DisplayList of objects in its view. Each of these is asked for a label to describe itself (by sending the object the *DescriptiveLabel* message). To determine where the arcs should be drawn each object is also asked for the value of its AnalysisLinks IV. An arc must be drawn between an object and every object in its AnalysisLinks which is also on the DisplayList of the AnalysisMap.

The AnalysisMap is responsible for arranging the nodes and arcs in a relatively pleasing way, for monitoring where the mouse is and what position its buttons are in, and, of course, for responding to all menus produced when its title bar is selected.

5.0 General network views

AnalysisMaps are quite general. Although they view statistical objects, they are not restricted to objects of statistical interest. An AnalysisMap views and provides an interactive interface to objects that are connected by AnalysisLinks and which will

respond to the left-button messages *Zoom*, *SetName*, and so on. The statistical and numerical operations are made available by querying the individual objects.

Consider the non-statistical content of an `AnalysisMap`. The links between objects are directional and can be determined from the objects themselves. As a general network view, the `AnalysisMap` responds to mouse selection in its title bar by producing a left-button or a middle-button menu as appropriate. Among the middle-button menu items there is the ability to narrow the view, widen the view, and so on. Similarly, a menu of items is produced if one of the objects being viewed is selected with the left mouse button. Every object is expected to be able to respond to the messages of those menu items. Similarly, each object is expected to produce a descriptive node label and a menu of items when it is selected with the middle mouse button.

These responsibilities can be abstracted to define a general network view that can be implemented as a class. This class, which we call `View`, is a general purpose tool for inspecting and altering a network of linked objects. An `AnalysisMap` is just a particular kind of `View`. The class `AnalysisMap` is said to be a specialization of the class `View`.

While all behaviours, like widening the view, creating a sub-view, and so on, are defined for `View`, the links to be followed between objects are not. The links are specified only to the extent that forward and backward links will exist. What kind of link is a forward (or backward) link will depend upon the kind of network being viewed. The forward links of an `AnalysisMap` are to be found on the `AnalysisLinks` instance variable of each object and the backward links on the object's `BackAnalysisLinks` IV.

This relationship between `View` and `AnalysisMap` is implemented in object-oriented programming by declaring `View` to be a parent, or super, class of `AnalysisMap`. All behaviours and instance variables of `View` are automatically behaviours and instance variables of `AnalysisMap`. The class `AnalysisMap` is said to inherit these from its parent class, `View`. (This idea of inheritance is used extensively in the definition of previous statistical objects as well so that methods like *MakeAnalysisLinks* and *Zoom* can be centrally located and shared by all statistical objects.) As a child of `View`,

AnalysisMap also has methods and IVs that are special to itself (e.g. an AnalysisMap determines its forward links by accessing the AnalysisLinks of each object).

A class can have many different children (and many different parents), each one specialized in different ways. In particular, View is specialized to allow other connections between statistical objects to be viewed (as discussed in Section 3). The children of View are shown in Figure 18.

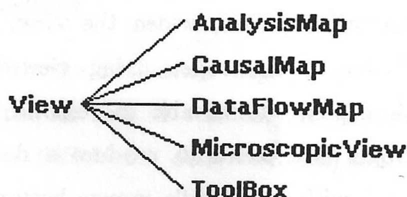

Figure 18. Inheritance from View

Like AnalysisMap, each child is a View specialized to follow certain links. A CausalMap traces the CausalLinks as the forward links which connect objects. A DataFlowMap goes to each object's RequiredIVs to trace the backward links between objects - forward links are then defined in terms of the backward links. A MicroscopicView was seen in Figure 6 of Section 4. MicroscopicViews follow an object's IVs as forward links. Unlike an AnalysisMap, none of these three Views allow the analyst to make or break links between objects in their view. Clearly, critical information could be lost if, for example, CausalLinks were broken.

A MicroscopicView also differs in that it cannot contain other MicroscopicViews. Nor can it contain more than one object and its IV values. A MicroscopicView is meant to be the bottommost view of any given object. Of course the value of any IV can also be zoomed in on (using the left-button menu) to get a MicroscopicView of it independent of the original object.

The ToolBox is a View quite different from the others. It views the available classes of statistical objects and, consequently, is a network organization of the statistical data structures that are available to the analyst (e.g. commands like REG or PLOT). A ToolBox can contain other ToolBoxes and links between the objects it views can be

manipulated. In this way, the analyst can personalize the organization (e.g. grouping tools (classes) into categories of analysis). This is a great advantage over a simple alphabetic listing of possible commands (as in S).

The many different kinds of `Views` discussed here afford the analyst the means to interactively investigate a variety of statistically meaningful connections between different statistical analysis objects. Should other connections become interesting to record and investigate in the future, a tool to monitor and perhaps manage them will be easily produced by specializing `View`.

6.0 Discussion

A statistical analysis is a rich structure having more connections between intermediate results and commands than the time order of events alone would indicate. Many of these connections will depend upon the actual problem being analysed and are therefore best left to the analyst to make (i.e. `AnalysisLinks`). This does not, however, preclude having software assist the analyst in making and managing these links. The design of the statistical objects discussed above and the `Views` which display their interconnections provide the analyst with a new graphical way to carry out a statistical analysis.

Moreover, the network model presented here does not require that statistical computations be organized at any higher level than present interactive statistical systems. It simply requires a shift to the richer structures of object-oriented programming. Data structures like vectors and arrays are easily given object-oriented representations. Statistical commands either become objects themselves (e.g. S's `reg` and `plot` become the object classes `REG` and `PLOT`), or they become methods of the data structures on which they operate (e.g. *LOG*). The former is likely the best choice for commands which produce a complex data structure, while the latter would probably be chosen for operators which either returned no data structure or returned one that is the same as its operand. (There may be considerable value in implementing some commands both ways. For example, the object `REG` could have a method called *PlotResidualsVsFit* which, when invoked, would instantiate a `PLOT` object with appropriate arguments. The plotting is implemented both as an individual class (`PLOT`) and as a method of another class (`REG`). The key result is that, when the `REG` object is selected in an `AnalysisMap`

with the middle-button, a menu will offer the possibility of plotting the residuals in this way. The analyst has a method available where she is likely to use it.)

Given this kind of behaviour, one is encouraged to consider different and possibly higher level organizations of statistical computations and data. In Oldford and Peters (1986, 1988), we suggest how the nodes in an `AnalysisMap` might also be taken to represent steps in a statistical analysis. Instead of the `REG` object representing the results from fitting a regression, it could represent the decision to model a vector of responses as a function of other vectors. It would respond to the message *DoLeastSquaresFit* by producing a new object called a `LeastSquaresFit` which would contain the results of the fit. (Of course, this new object would be linked via analysis and causal links to the `REG` object and would appear in the `AnalysisMap`.) `REG` could have other messages like *DoRobustFit* and *PlotData*, representing different choices at that step in the analysis. These would make life easier on the experienced analyst and, if coupled with good on-line documentation, would provide a less experienced analyst a modicum of guidance in unfamiliar territory (see Oldford and Peters, 1988, for further discussion).

Finally, by using a tangible model of a statistical analysis, patterns should become more apparent. These may have strategic import, either to record and study, perhaps to repeat, or to avoid altogether. Before they can be affected, they must be recognized.

7.0 References

Becker, R.A. and J.M. Chambers. (1986). "Auditing of Data Analyses", *Siam Journal for Scientific and Statistical Computing*, (to appear).

McDonald, J.A. and J. Pedersen (1986). "Computing Environments for Data Analysis Part 3: Programming Environments", *Siam Journal on Scientific and Statistical Computing*, (to appear).

Oldford, R.W. and S.C. Peters (1986). "Data Analysis Networks in DINDE", *Proc. of the ASA: Stat. Comp. Section*, pp. 19 - 24.

7.0 References

Oldford, R.W. and S.C. Peters (1988). "DINDE: Towards more sophisticated software environments for Statistics", *Siam Journal on Scientific and Statistical Computing, 9*, pp. 191 - 211.

Stefik, M. and D.G. Bobrow (1985). "Object-Oriented Programming: Themes and Variations", *The AI Magazine, 5*, pp. 40-62.

Stuetzle, W. (1987). "Plot Windows", *JASA*, *82*, pp. 466-475.

Teitelman, W. (1972). "Automated programmering - The programmer's assistant", *AFIPS Conference Proceedings, 41*, pp. 917-921.

Teitelman, W. (1977). "A Display Oriented Programmer's Assistant", *Proceedings of the Fifth Joint Conference on Artificial Intelligence*, Cambridge Massachusetts, pp. 917-921.

Xerox (1985). *Interlisp-D Reference Manual Volume II: Environment*, Xerox Corporation.

Cover By Anne Sprott