Keeping Statistics Alive in Documents[1]

Günther Sawitzki

StatLab Heidelberg, Im Neuenheimer Feld 294, D 69120 Heidelberg
<gs@statlab.uni-heidelberg.de>

Abstract:

We identify some of the requirements for document integration of software components in statistical computing, and try to give a general idea how to cope with them in an implementation.

1. Introduction

Stimulated by developments in data analysis, we have seen drastic changes in the way we use statistics over the last decades. With advanced technical possibilities and computational based methods, dynamic aspects have become more important: interactive methods are common in data analysis now; simulation supported methods have first class importance; "experimental statistics" is by now an indispensable part in statistical research and method development.

If it comes to disseminating methods and results, we are largely lagging behind. Though there are good examples how to enhance paper based publications by software (adding a CD, say) or for making use of the web-based facilities, most of the papers and publications are "dead", even where the contents would strongly suggest a live presentation. The question is which technologies can be used to create live documents, and which special conditions should be kept in mind. We will discuss the use of component technology in document generation for statistical computing. Scatterplot brushing and linked windows can serve as well known examples of what we want to cover. These are basic techniques in interactive data analysis. In a paper document, we are restricted to static pictures or (long) verbal descriptions. A live document should preserve interactivity and global consistency of the linking.

Ad hoc solutions are possible: we can produce special purpose software, possibly linking to an underlying pre-existing "compute machine" to drive a multi-media document on a CD, say. ActivStats (Velleman et al., 1996) is an excellent example of this approach. But we want to go beyond this and ask for technologies to create compound documents, including the typical, possibly interactive elements we need in statistics. We want these technologies to provide the

[1] This paper was written during a visit to Sonderforschungsbereich 373 ("Quantifikation und Simulation ökonomischer Prozesse" at Humboldt-Universität zu Berlin, supported by the Deutsche Forschungsgemeinschaft. Thanks to W. Härdle and the members of his group for hospitality and discussions. Special thanks to W. Härdle for granting access to his personal Oberon variant. Presented at SFB 373, Berlin, Sep. 14th, 1999. Part of this material was presented at the 52th Session of the, ISI Helsinki 99.

same flexibility and ease for interactive elements as nowadays editors do for preparing (non-interactive) papers and reports.

Component technology in principle allows composing documents for statistical computing, such as reports and status charts, as well as other documents like methodological papers or teaching material, from predefined reusable components. The advantage is on the one side to open new realms for documents with dynamic character, to be used for example for delivering up-to-date information (if the dynamics are controlled by the state of the data) or allowing true interactivity (when the dynamics are controlled by the user). The other advantage of component technology is to put document generation in the hands of the author and to give him the freedom to compose the document using available components, relieving him from the dependency on ad hoc solutions.

In general document processing, support for component technology is already available in a limited form. Editors may allow extensions implemented for example as dynamic link libraries (DLLs) and presented as ActiveX or OpenDoc elements to be embedded in a text. Likewise, HTML allows imbedding applets and scripts to provide dynamic facilities, PDF allows imbedding JavaScripts.

The kind of documents needed in statistical computing generally are data based, and the link to the underlying data should be preserved and reflected in the documents. In statistical data analysis for example, this leads to linked dynamic plots, and a state-of-the-art implementation of documents should preserve linking and dynamical characteristics, while preserving the performance needed for statistical computing.

We will identify some of the requirements for document integration of software components in statistical computing, and try to give a general idea how to cope with them in an implementation. Our documents used as an example are taken from research papers or teaching materials, but the discussion applies to other material such as reports or status summaries as well. Most examples focus on graphical output. Adequate handling of statistical displays seems to be the most challenging aspect, but the general discussion applies to other displays such as tables or summary statistics as well.

We start with an illustration of what we mean by an "integrated document", and give an example of what we mean by "software components" in statistical computing. In the main part, we discuss the necessary requirements on tools for making documents. We will focus on three general features of documents:

- Documents have well defined contents which are maintained in a reliable way. *Persistence must be supported.* Document contents as well as dynamic linking must be preserved if documents are stored or communicated.
- Documents are structured internally and each part has a context. *Structure and context re-*

lations must be supported. Components should be sensitive to their context and adapt to the structure and context of the embedding document, allowing pre-defined components to be used in an efficient and flexible way.

- Documents may be communicated. *Sharing of documents and data must be supported.* This means taking account of problems possibly which may arise from duplication of information, partial or delayed access, or different user environments.

A document is always used in an environment, and any user will have his or her preferred environment. These choices should be respected. In a final part, we discuss merging of documents and components in the user environment, taking this into account.

2. Integrated Documents

Figure 1 illustrates what we mean by an integrated document. The paper (Sawitzki 1994) originated as a loose-leaf collection of notes, used for consulting. Figure 1 shows elements from an interactive version. The first thing to note is a seamless integration of graphics in an imbedding text. The possible interaction cannot be seen in the printed version. In the interactive version, the graphical elements are live - even when imbedded in the text. The reader can query the data values - clicking on a plot will open a window with the data set. Dragging your own data from a spreadsheet or some other source onto the plot will change it to show your data. Copying the plot to your own document lets you integrate the plot - with all interactivity preserved - into your own document.

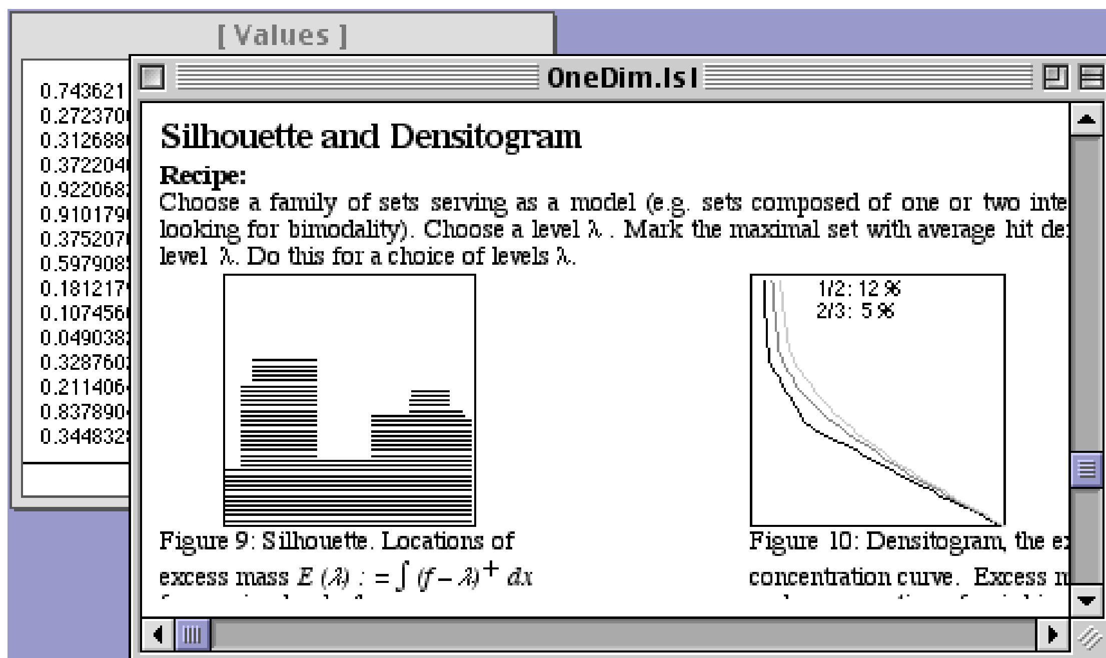

Figure 1: *an excerpt from an integrated document.See interactive version.*

While drag&drop is a mere convenience when used with static data, its main effect so far could even be achieved in a command-based environment, or by linking a plot to some “form” which allows data entry (a spreadsheet mask, for example). True interactivity, and true dynamic integration comes into play if you apply it to other elements. For instance, the illustrations allow the user to draw a distribution by hand, sample data from this distribution, and show the sample in any of the displays. If the distribution is changed, a new sample is drawn. Linking a time signal from a clock would give a running simulation. See the “Readme” file which comes with (Sawitzki 1994) to try these examples, or see the video (Sawitzki 2000).

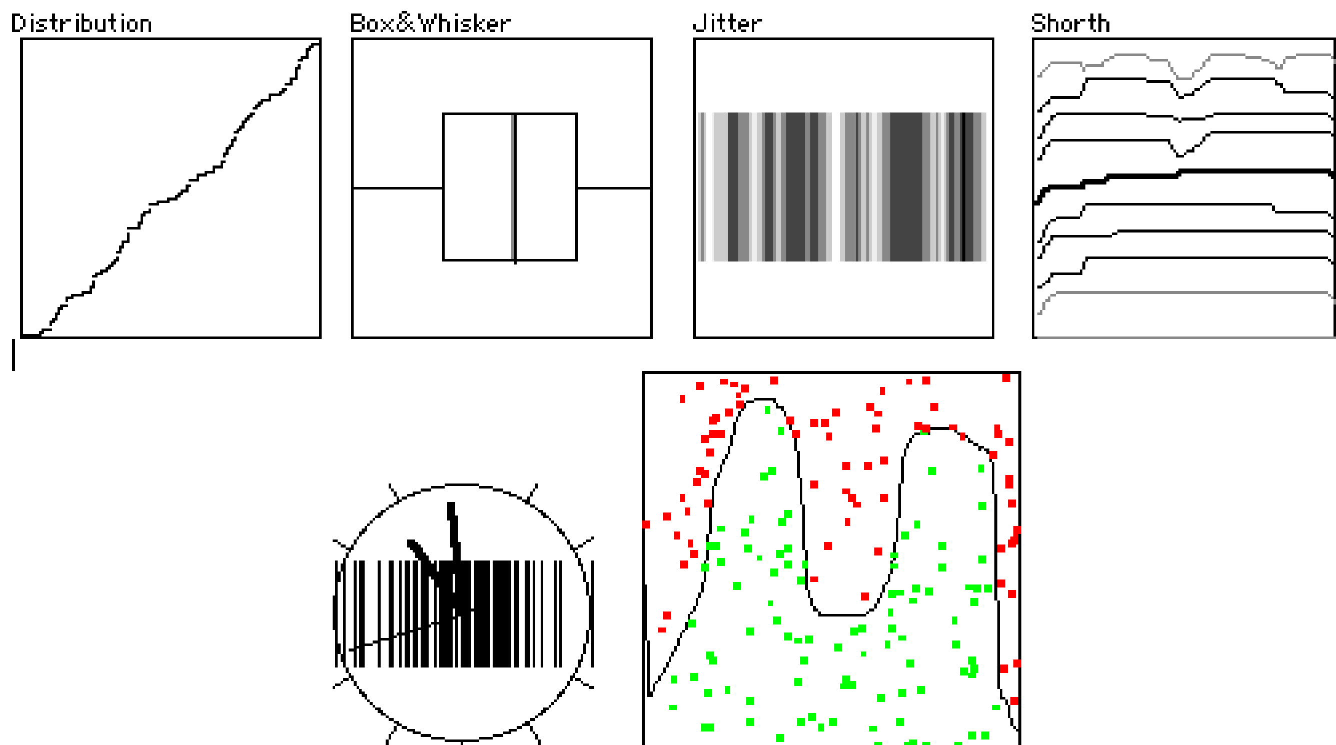

Figure 2: *Integrated Documents. Interactivity, e.g. drag&drop data to displays, mouse input for distribution, linking are supported. See interactive version.*

Besides the by now familiar integration of texts and graphics, the integrated document preserves the dynamics of the graphics. This was a new feature at the time the example was published. Nowadays, active elements or applets are readily available ways to implement this. What goes still beyond active elements or applets are dynamically reconfigurable links. For example dragging a data set of a display in Figure 2 makes it show that data set. Dragging a distribution will link the display to a random number generator with that distribution, and random samples can be shown. If additionally the clock is dragged to the distribution, the clock pulse will be linked to a random number generator and the result you see is a real time simulation.

Let us have a short look behind the scenes. The technique to achieve this is the well-known model-view-controller separation, already known since Smalltalk and quite common in Java: each display element is actually a *view* on an underlying *model.* Two views on the same model may be different, but they always will be consistent as far as the data represented in the model are concerned. If data or a data generating object are dragged on top of a display, their model replaces the recipient model, leading to consistent linking. If a model changes, a *message* is broadcast through display space, triggering an update. This is all the magic which is behind the dynamic linking.

3. Software Components

The example for software components is taken from a technical research paper. For this example will need some statistical background. We consider some observed input, which may be a noisy signal:

$$\textit{Input } Y = \textit{signal} + \textit{noise}$$

Working in discrete time, we have the input signal observed at n time points. To make life simpler, we consider the canonical model

$$Y \textit{ random vector in } \mathbb{R}^n$$

$$Y \sim N(\mu, \sigma^2) \textit{ with } \mu \in \mathbb{R}^n, \textit{ n large.}$$

Here μ is our signal. If we have copies of the input, we could average to recover the signal. But Stein taught us: the arithmetic mean is not an admissible estimator for μ. Use shrinkage! The argument still holds if we just have one copy: in particular, Y is not an admissible estimator for μ.

Applying Stein shrinkage in general is not helpful in signal recovery. The important observation of (Beran and Dümbgen 1998) was: Stein's argument applies to all representations of Y, not just to the given one. So instead of working with the original input Y, they may use some "good" orthogonal basis for $\mathbb{R}^n$. Stein shrinkage is applied to the transformed signal, and then the inverse transformation is applied. "No transformation", the identity, is but a special case of this setting. The additional idea to allow varying shrinkage factors opened access to the potential inherent in James-Stein estimators.

We can get a full picture of our data by a series of displays (Figure 3). None of these displays is problem specific - all of them can be generated by "off the shelf" software.

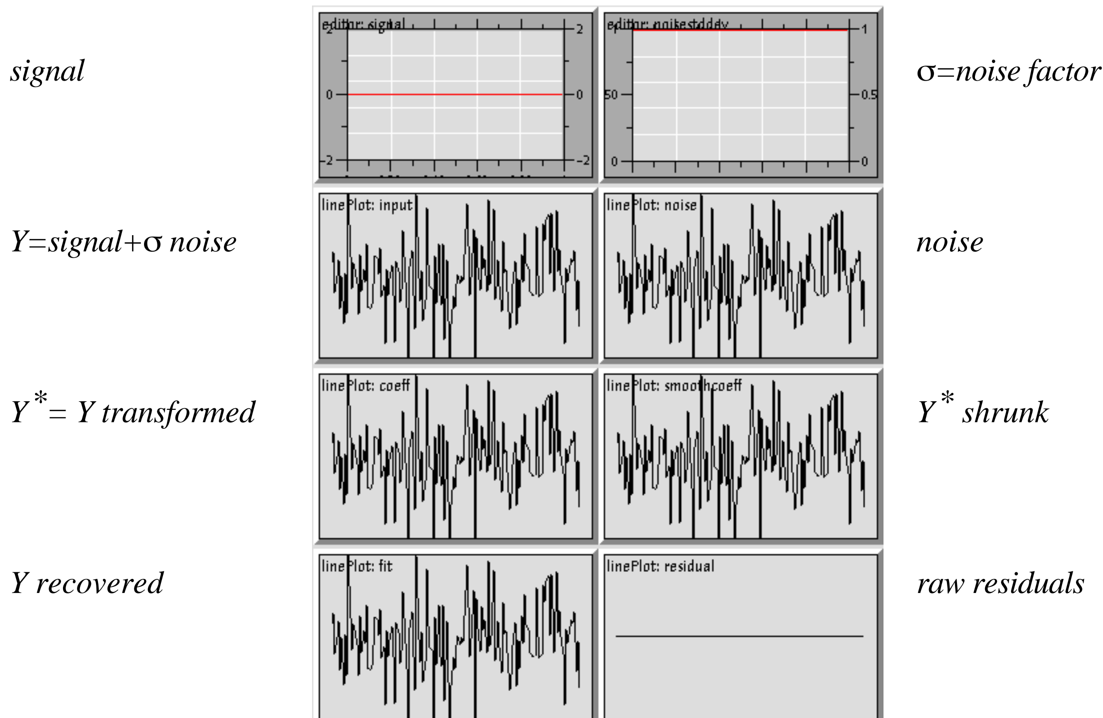

Figure 3: *Views on data for signal recovery (no transformation applied, no shrinkage)*

If we have properly linked windows, we already have a dynamic model. We can modify signal or noise, or draw repeated samples. But with classical software, we would have to modify our programs to apply the Beran/Dümbgen idea for Stein shrinkage. With software components, we can modify or replace individual components. While initially (no transformation) the data are copied over to go from Y to Y^* resp. from (Y^* shrunk) to (Y recovered), we can replace the copy operation by a Fourier transformation resp. its inverse (usually something which can be taken off the shelf in a statistical system). In a component system the links should stay intact. Replacing the copy operation from (Y transformed) to (Y^* shrunk) by a variant of Stein shrinkage (e.g. Dümbgen’s isotonic shrinkage), we get a full simulation environment for REACT estimators. (The acronym REACT stands for risk estimation and adaptation after coordinate transformation).
As a convenience, we can add a control panel to select or replace components by button control (Figure 4).

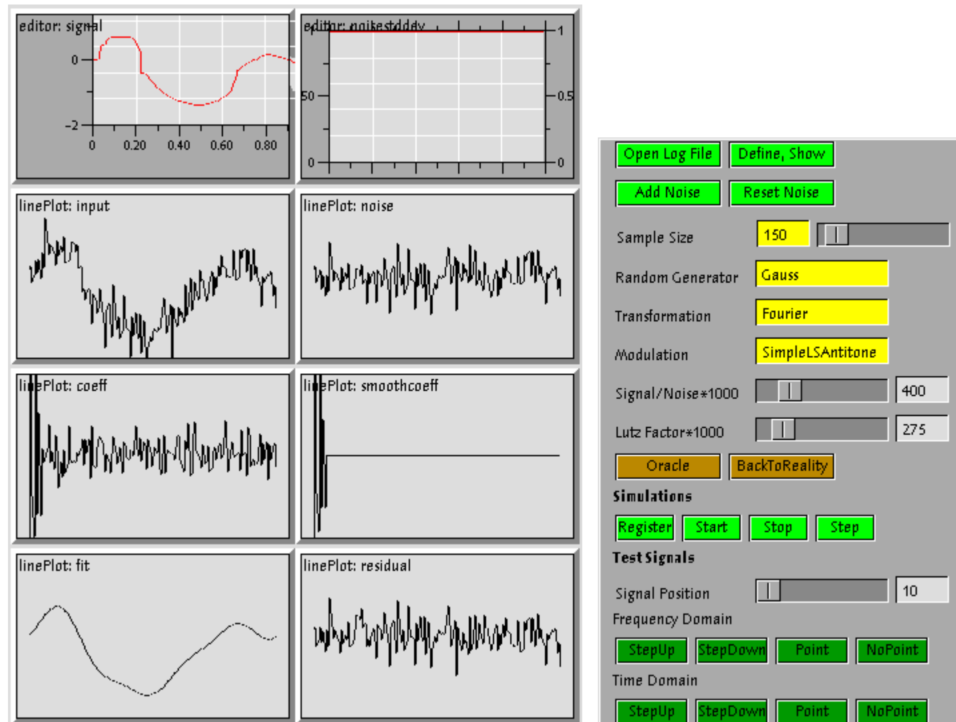

Figure 4: *Simulation for REACT. Only minimal custom components need to be added.*

Re-usable components for display *...for control: just re-used*
Plugged-in generic ON transformation *software components !*
Plugged-in custom Stein shrinkage (Dümbgen, 1998)

Instead of relying on ad hoc solutions, this example makes use of integration and modification of pre-existing re-usable software components. Given the speed and interactivity of the Voyager environment (Sawitzki, 1996) used here this is an effective tool for Experimental Statistics in the sense of (Beran, 2000).

4. Making Documents

What does it take to make a document[2]? Documents seem so familiar to everyone, but we do not know if we have a common understanding of what we mean by a document. Here are some features which we will assume in our discussion:

- Documents are self contained
- Documents may be
  - stored & retrieved in a reliable way
  - passed to others, or possibly shared
  - authorized and authenticated
- Documents may be
  - structured internally
  - linked externally

[2] from docere, "to teach".

- Documents can be built re-using components

The first properties will be generally accepted. The last one is a late addition, coming with nowadays Taylorization. Documents have lost their originality. We use text fragments and predefined clauses to write a text in an office, and we expect the same to hold more generally. Tools to generate and manage integrated texts should support possibilities to manage and reuse text fragments for integrated documents as for plain texts.

We will come back to these key features of documents as we discuss selected aspects of document production.

4.1 Persistence

Documents are self contained and documents may be stored & retrieved in a reliable way. In technical terms, this means that the document information is *persistent*. To understand the implications, we need some case studies and must try to understand the specific needs of users.

In our first example, the paper on diagnostic plots, the user may be a reader. The "document" is the paper and all components are contained in the document; it can be stored by "dumping" all information. Components may be changed, e.g. when the reader chooses a different distribution. The appropriate action is not obvious if components are changed. In the paper, a way out was to provide different modes of usage. If the document is read in browser mode, the original version is restored. If opened in editor mode, the most recent state is stored, and the document will open as the user has left it.

For the same example, the user may be a client, for example someone using the plots to generate a report. The scope of the document is now user defined; components may be extracted and/or data may be used which are external. A poor way out is to cache the data with the display and to restore the cached information when the document is reopened. In general this way out will not be acceptable since it duplicates information, but does not guarantee consistency. A better solution will be discussed later on (section 4.3).

The second example, the REACT plot, with the author being the user, reveals more difficulties. Displays and controls may be separate documents; components may be extracted and controls may be extracted or removed. Status information may or may not be adapted to the current state. A way out may be to store all critical information with controls. When reopening, a control script can run to recover the documents. Sometimes this may be sufficient, but it is a poor way out and is not generally acceptable.

The dilemma is that storing too much information is uneconomical, may reveal critical information which is not public and may lead to inconsistency due to duplicate information while storing too little information may omit critical information and may lead to inconsistency due to lack of information. Unfortunately finding the right amount of information is not trivial.

From a general point of view, the system state is defined by the state of its objects and the dependencies between these. This can be represented by a graph with the objects as nodes and the dependencies as links. The document state is defined by the visible objects in the document. So the abstract task is to find a minimal subgraph to reconstruct the document and to store the state of this subgraph.

Unfortunately there is more than one source for dependencies and the system graph is not evident. We can study this in the REACT example. Starting with the visible objects, the view-model-controller design lets us ask additional invisible objects for the models and an implicit dependency between view and underlying model. But in this example, we change or redefine the transformations which act as pipelines between models, thus leading to more “hidden” dependencies.

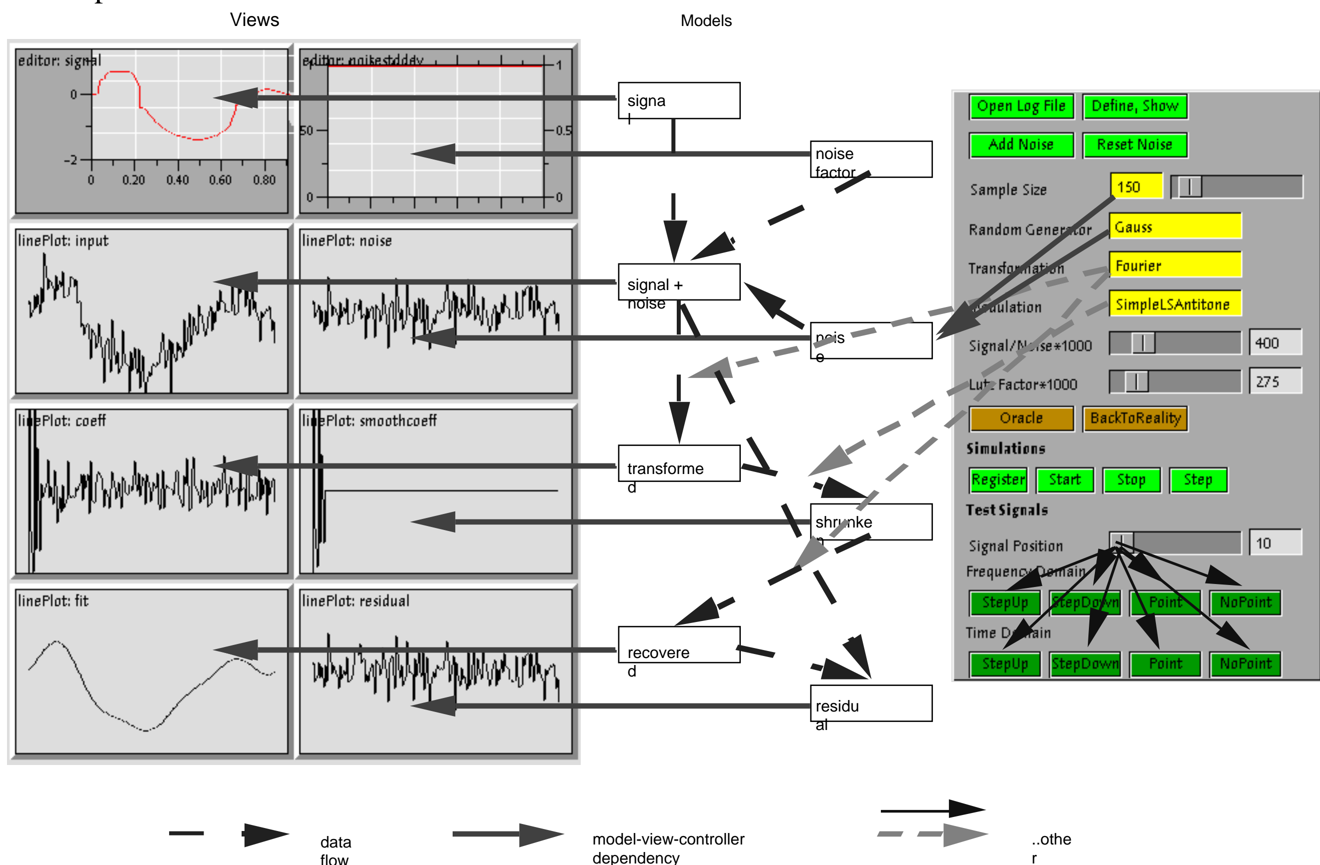

Figure 5: *Dependency Graph for REACT (partial). Various sources of dependency contribute to form a complex graph.*

The dependency graph is not explicit and contains contributions from various sources. Each of these sources has its ways to recover the contributed dependencies. The main contributions come from these sources:

- Control flow. For example, a new signal or a new noise sample triggers evaluation of new input in REACT which propagates until finally a new estimator triggers evaluation of new residuals. Control flow dependencies can be recovered by classical control flow analysis.

- Data flow. For example, residuals depend directly on input and estimator. Data flow dependencies can be recovered by classical data flow analysis (at least as long as aliasing problems do not interfere).

From the design we get
- Model-View-Controller dependencies. These are explicit (at least if the system is well-designed). For example, the random graph of the random noise gives a view on a simulation sample model, the sample size being controlled by a slider.
- Transformation dependencies. May be inferred from the syntax tree (if available). For example, the modulation in our example gives an additional dependency on the choice of the modulation type.

In an object oriented system we have additionally
- Object inheritance. May be inferred from symbol table (if available). For example the editable function plots for signal and noise factor are descendants from a general plot class used in the other displays, adding additional support for free hand drawing.

If a message passing system is used we have to add
- Message dependencies. These need to be traced dynamically (sometimes). For example a change of the entry field for the sample size must be signaled to the underlying model, passed on to the sampling model where it triggers a new sample, leading to a new message which must be propagated until ultimately all affected graphs are updated.

In total, to allow reconstruction of the dependency graph needs careful design and careful choice of development environment to keep the task feasible. Once the graph is reconstructed, finding a sufficient subgraph to recover the model can be done by standard algorithms, and storage and recovery of graphs can use standard computing techniques (e.g. pointer swizzling, see Griesemer et al. 1991).

4.2 Context Sensitive Re-Use of Components

Reusing components is straightforward if the components are self contained. Managing component dependencies is related to the problems discussed above, although in general much simpler than the persistency problem.

The main challenge is to take into account context information. As all graphical environments, documents have a context structure. In a graphical environment, this is of central importance, while in a classical environment (as a result of much work) it has been largely eliminated or formalized in scope constraints (like proper definitions of "calling frames" or the introduction of block structures). This context sensitivity is one of the main differences between graphical user interfaces and command line interfaces. In command line interfaces, we can avoid context sensitivity or restrict it by formal rules. In graphical interfaces we do want to support the context sensitivity, and this support becomes a central design feature.

As an example (Figure 6), take an excerpt from an introductory course on statistics, imple-

mented using Voyager (Sawitzki, 1996).

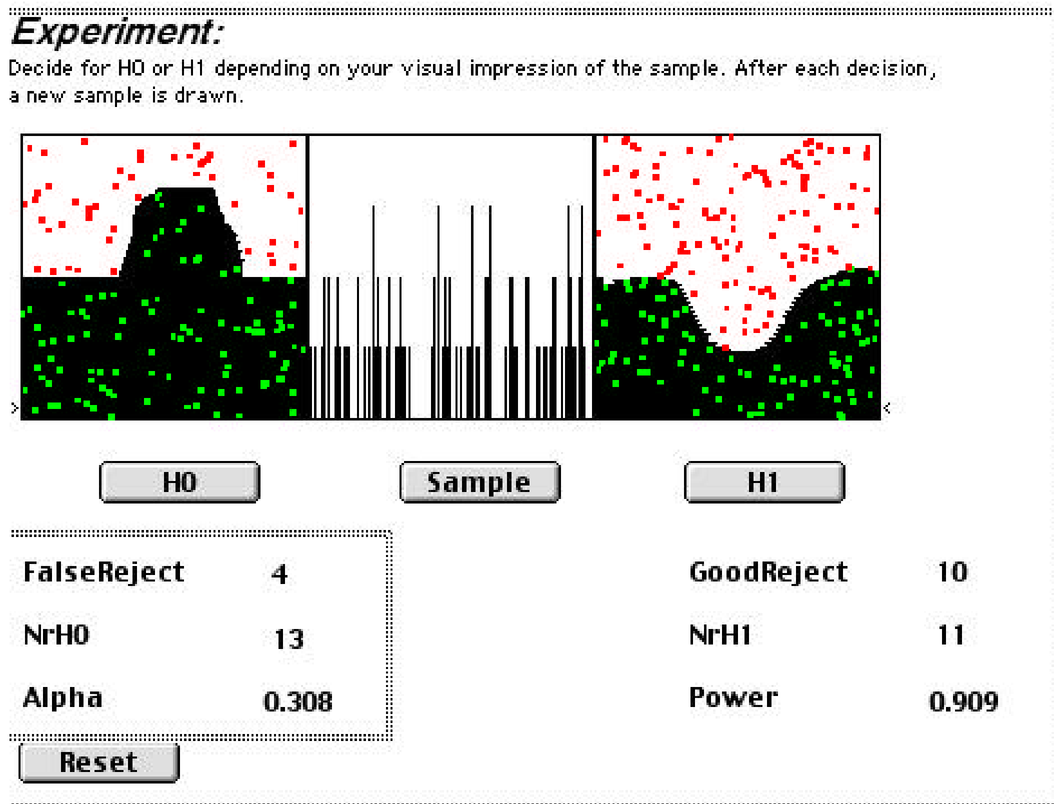

Figure 6: *Generic "Sample" button with context-defined action. Two context groups are outlined in this figure for illustration.*

In this example, a pre-defined button labelled "Sample" is used. Sample should activate drawing a new random sample. From the author's perspective "Sample" is a generic action. It should get its specific meaning from context. The problem is to determine the context. In a classical environment, we use unique identifications for a target (e.g. by names or access paths), or we use rules of scope. For document construction with graphical environments, we have to introduce context sensitivity.

A practical solution which has proved to be sufficient so far is to introduce invisible structuring elements to define scopes (two examples are outlined in grey in figure 6). The "Sample" button uses a local broadcast to call for a new sample; this will affect all components in the button's context, marked here by the enclosing dotted line, but no components outside. If moved to a different context, the action will change correspondingly, without re-programming. Similarly, if the user selects to accept the hypothesis (clicking H0) in this example or rejects it (clicking H1), the corresponding counters in this context are increased, and a message is forwarded to the "Sample" button, yielding a new sample.

In rare cases where context identification is not sufficient, we fall back to classical implementations and use (unique) names and references by name.

4.3 Sharing Documents and Data

Sharing documents and data goes beyond transporting documents or data. We have to ask: what does the recipient need to make use of the document or data? Additionally, we have to ask what we can assume about the availability of

- Computing resources
- Additional software
- Network facilities
- Data access

In an ideal world, no precautions are needed. But if the recipient has poor connectivity, or access is critical, we should avoid network access as far as possible. In some cases, data may even be unavailable. We should use fall back solutions for these cases. We give some details of solutions which have been helpful to meet unfriendly conditions.

4.3.1 Implementation Details: Caching Data Structure

We use caching data structures for example to implement a vector data structure

```
Vector = RECORD
      cache:       cache list
      buffers:     access paths, with buffer cache
END;
```

So we use a two level caching. Besides the vector cache, the vector is composed of buffers, and each buffer has its own cache. For a data element of vector type we calculate the mean for example by an implementation which follows these steps:

```
try to find mean in vector cache
on failure: add mean to vector cache from buffer caches
      …try to find mean in buffer caches
      …on failure: add mean to buffer caches
```

For details, see the documentation in <http://statlab.uni-heidelberg.de/projects/voyager/>. Using this caching data structure with buffers is a very helpful implementation if we go to where communication may be most critical: to distributed computing. All buffer specific calculations can be distributed in a natural way, and only the derived (possibly cached) summary statistics need to be exchanged.

4.3.2 Implementation Details: Object Stamping

Objects have various access stamps, for example as in

```
Object = RECORD
      modtime:     a time key
      key:         a checksum, for example
      .
      .
      .
END;
```

```
obj:  Object;
```

To re-access data for an objects of this type on an expensive channel, use

```
Get(obj.modtime); Get(obj.key);
IF (obj.modtime   savedmodtime) OR ((obj.key   savedkey) THEN
      Get(obj.data)
ELSE
      obj.data:=saveddata
END;
```

Again, for details, see the documentation in <http://statlab.uni–heidelberg.de/projects/voyager/>.

4.3.3 Implementation Details: Hot and Warm Links

If we do not have a guaranteed environment on the recipient's side, or access is not guaranteed, we have to prepare for outdated objects/displays. A convenient way to handle these can be seen in Paul Velleman's DataDesk (Velleman 1984, Figure 7). A plot which is not consistent with the current data state is marked, and alternative actions are offered.

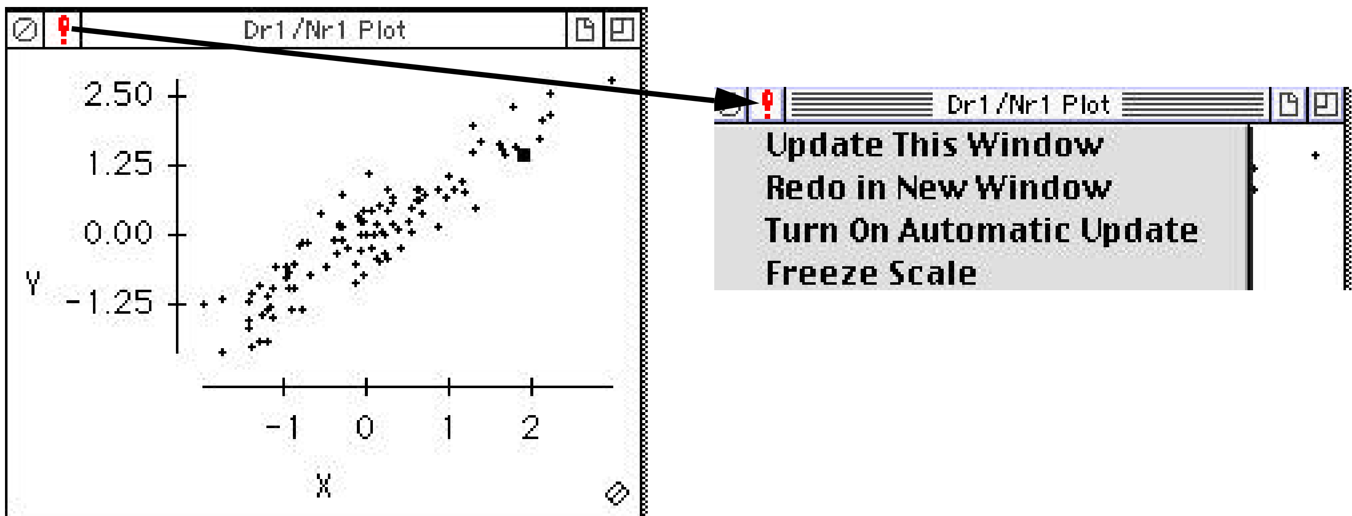

Figure 7: *Handling of loose links in Data Desk. Outdated plots are marked. The visual mark acts as a hyperlink to a selection menu offering appropriate actions.*

Out-of-date is an object attribute in DataDesk. But different links may have differing reliabilities, and the appropriate action upon an update event may differ. So in a more general context, it should be a link attribute

```
Link = POINTER TO RECORD
      obj:              link target
      link: Link        chain
      status: SET disposition and status flags
END;
```

5. User Environment

Each user has his or her preferred environment, including printed copy to read and scribble on while travelling lightly. As developers, we have to accept these choices and accept user preferences. Software should be adapted to the working environment in which it is used. There are several levels of merging with the user environment. The main examples illustrate these. Both examples in fact make use of a new operating system to provide their base, the Oberon operating system (Wirth, N. and Gutknecht, J. 1992). In the REACT example, the screen shots are taken from an Oberon implementation which runs on top of a Macintosh operating system. The screenshots will look nearly identical when taken from a Windows machine, or from UNIX, or from a native Oberon machine.

The diagnostic plots example is taken using a different implementation of Oberon. Again, this implementation can run on top of other operating systems, such as Mac or Windows. But this one has the specific look and feel of the host operating system: it looks like a Macintosh program on the Mac, and has the Windows look and feel when running on a Windows machine. For details, see (Sawitzki, 1996).

The next step is true integration. One way would be to port the software, the other is to use a portable base and keep the software unchanged. Using recent developments (Zeller, 1999) the components can run truly integrated in a Windows environment (as ActiveX elements), and can thus be integrated in software like Excel, Word, or any software that supports ActiveX, or in a browser like Netscape (supported by a plug in).

The implementation uses three layers

Voyager
Oberon System
Host operating system

(or two, if native Oberon is used). The host operating system is what the user has chosen. Oberon is the operating system we are actually working with, and Voyager provides the components for statistical computing.

In detail, there is a long list of implementations, as for example

Voyager				
Oberon system	System 3 or BlackBox	System 3 or BlackBox	System 3 - PlugIn	System 3
Host operating system	Macintosh	Windows	Windows & Netscape	Linux

In our example, we used BlackBox for the diagnostic example, and System 3 for the REACT plots.

6. From Legacy Texts to Integrated Documents

The core of our considerations is a matter of architecture, not necessarily of implementation. Similar considerations apply, whatever technology is used for implementation. Of course implementation is simple if a component based technology is used, like Oberon. But similar results can be achieved with other technologies (although slightly more work may be necessary).

We already discused editor extensions using ActiveX or OpenDoc to imbed dynamic elements in "classical" text documents, and dynamic link libraries (DLLs) allow this to be done in a flexible way. This is actually the way which is used to imbed the Oberon components seamlessly in Windows, as discussed in section 5. Using Plug-In-Oberon allows Oberon components to be used as ActiveX elements in WORD, say - no changes or re-compilation being needed. Other systems, like e.g. XploRe, are actively supporting this path to provide services via DLLs (XploRe 1999). DLLs, however, have the drawback that they are not portable. Even if you stay on the same machine and switch from Windows to Linux, say, they may become unusable.

PDF with active links or HTML in combination with applets are a related possibility. This approach lacks editor support, but gains from access to JavaScript (for PDF) or even Java applets (for HTML), for example, which provide a base for modular, platform independent programming. A collection of statistical applets is available from (Heinecke & Köpcke, 1998).

On the software bottom line, DLLs and Java applets are two technologies which can help developing integrated documents. Of course the details, as discussed in section 4, have to be taken care of by the programmer if these technologies are used.

Another question is how to move form legacy texts to integrated documents. This is related to the problems of multi-media production for statistical texts. A particular challenge is to take up the TeX legacy. While still not matching the quality achievable in traditional typesetting, it is marking a level of mathematical typesetting which we do not want to miss. Starting from legacy texts in TeX-format for example, part of this process can be automized via tagging TeX files.

An intermediate step is to supply hot links, which actually serve as command buttons to call additional software (Müller 1998).

While DLLs or applets are ways to provide a computational base, and tagging and preprocessing can help paving the path for legacy material, the issues discussed above show how to handle other problems related to integrated documents. The main work still is the design of integrated documents. Making proper use of these technologies requires investigation, critical discussion, and imagination.

Literature:

Beran, R.; Dümbgen, L. (1998): Modulation of Estimators and Confidence Sets. Ann Statist. 26, 1826-1856..

Beran, R. (2000): The Rise of Experimental Statistics. To appear in: Festschrift for D. Fraser.

Dümbgen, L. (1998): Isotonic shrinkage. Personal Communication.

Griesemer,R.; C. Pfister (ed.), B. Heeb, and J. Templ: On the Linearization of Graphs and Writing Symbol Files. Technical Report 156, ETH Zürich, Institute of Computer Systems, March 1991.

Heinecke, A; Wolfgang Köpcke, W. (1998±): JUMBO: Java enhanced material for biometry. Münster. <http://medweb.uni–muenster.de/institute/imib/lehre/skripte/biomathe/jumbo.html>.

Müller, M.: Computer-assisted Statistics Teaching in Network Environments. COMPSTAT'98 Proceedings, Bristol, UK.

Sawitzki, G. (1994): Diagnostic Plots for One-Dimensional Data. in: P.Dirschedl & R.Ostermann (eds.) Computational Statistics. Heidelberg: Physica, ISBN 3-7908-0813-X, 234–258. <http://statlab.uni–heidelberg.de/projects/onedim/>.

Sawitzki, G. (1996): Extensible Statistical Software: On a Voyage to Oberon. Journal of Computational and Graphical Statistics Vol. 5, 263–283. <http://statlab.uni–heidelberg.de/ projects/voyager/>.

Sawitzki, G. (2000): Integrated Documents. Video. <http://statlab.uni–heidelberg.de/projects/onedim/www/onedim.asx>.

Velleman, P. et al. (1996±): ActivStats. Addison Wesley Longman. <http://www.datadesk.com/ActivStats/>.

Velleman, P. (1984): Data Desk. Data Descriptions Inc. <http://www.datadesk.com/>.

Wirth, N. and Gutknecht, J. (1992), Project Oberon. Reading: Addison-Wesley. ISBN 0-201-54428-8

XploRe (1999): <http://www.xplore-stat.de/tutorial/programming/dll>

Zeller, E. (1999): Fine-grain Integration of Oberon into Windows using Pluggable Objects. <http://www.cs.inf.ethz.ch/~zeller/plugin.html>.

Figure 7 has been prepared using DataDesk (Velleman 1984). All other screen images have been prepared using Voyager (Sawitzki 1996).