Software Components and Document Integration for Statistical Computing

Günther Sawitzki
StatLab Heidelberg
Im Neuenheimer Feld 294
D 69120 Heidelberg
gs@statlab.uni-heidelberg.de

1. Documents and Software Components

Over the recent years, software component technologies have seen major progress. Applets or plug-ins are by now familiar forms of software components. For statistical computing however, so far they have played only a minor role, such as providing interface elements or demonstration tools with limited functionality. Their potential reaches much further and still needs to be explored.

Software components usually are small building blocks, in contrast to more or less monolithic applications. Software components can be dynamically integrated into larger applications, allowing customization for certain tasks. Going beyond this, components can be combined to build complete systems, sometimes making monolithic applications obsolete.

Voyager is an example of a system for statistical computing built from components. For an introduction to Voyager, see (Sawitzki 1996). Voyager is based on Oberon from ETH Zürich (Wirth and Gutknecht 1992). Oberon comprises both an operating system and a programming language. We report about some experiences with Voyager, in particular the possibilities to build self-contained integrated documents. The examples are taken from teaching material, but the experiences immediately generalize to other applications such as report generation or data monitoring.

Documents are self contained and store their information consistently, even if exchanged or edited. A key property of documents is that they can be exchanged, passing complete and consistent information. For computing, we have to find reasonable limits within which we follow this metaphor.

As a first step, we have to

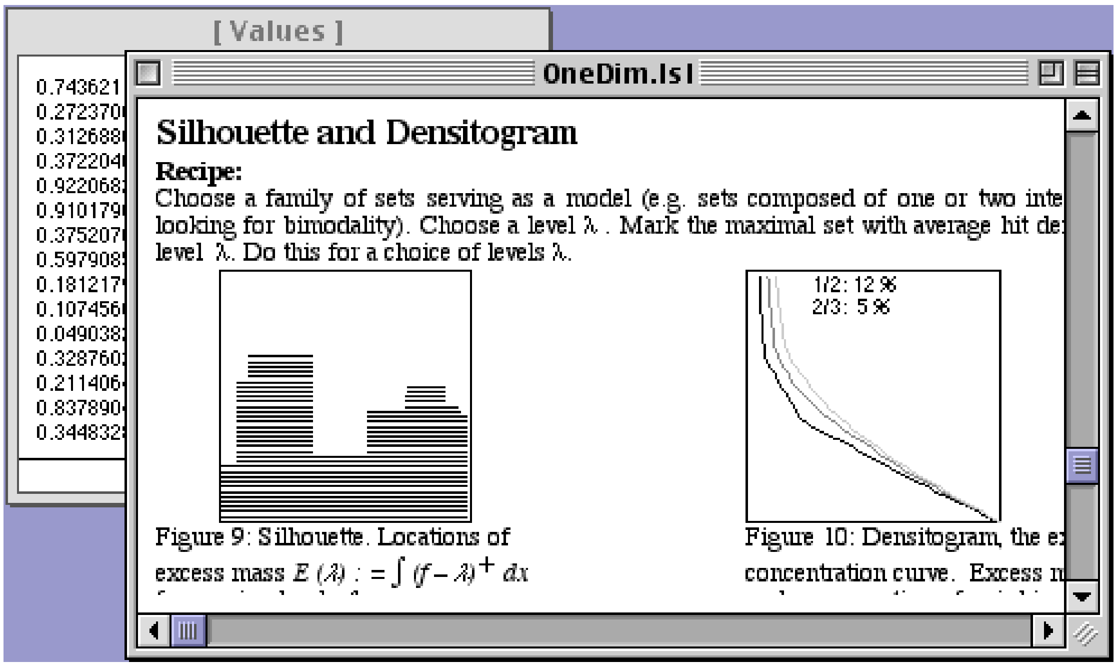
2. Embedded Views

Example 1 is from a paper on diagnostic plots for distributions on a line. 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: the graphical elements are life - 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.

The main challenge is persistency in an interactive environment. While for a report, we would like the content kept immutable, for many other purposes we want documents which can be changeable, and these changes should be persistent. In an Oberon environment, we can have document modes allowing both. In other environments, persistency of changes may be a problem. PDF, for example, would allow annotations, but no real changes of the document data. Java, to name another example, does not even have a true document model, and problems are abundant.

A “static” implementation of this example would only need an environment where graphics can be imbedded in text and used as hypertext elements - a simple PDF document would do. Seamless access to data input and output needs more support. In the version available on the net, this has been achieved by using an Oberon implementation (BlackBox) running as a framework which is hosting Voyager. In a true Oberon system, this framework is not necessary, since the Oberon itself supports these functions. It also solves the question of network access: Oberon accepts URLs as path indicators, so there is no difference between local and remote access.



0.743621
0.272370
0.312688
0.372204
0.922068
0.910179
0.375207
0.597908
0.181217
0.107456
0.049038
0.328760
0.211406
0.837890
0.344832

Silhouette and Densitogram

Recipe:
Choose a family of sets serving as a model (e.g. sets composed of one or two inte
looking for bimodality). Choose a level λ . Mark the maximal set with average hit de
level λ. Do this for a choice of levels λ.

Figure 9: Silhouette. Locations of excess mass $E(\lambda) := \int (f-\lambda)^{+} dx$

Figure 10: Densitogram, the e concentration curve. Excess n

Figure 1. Embedded Views. The document contains "life" components.

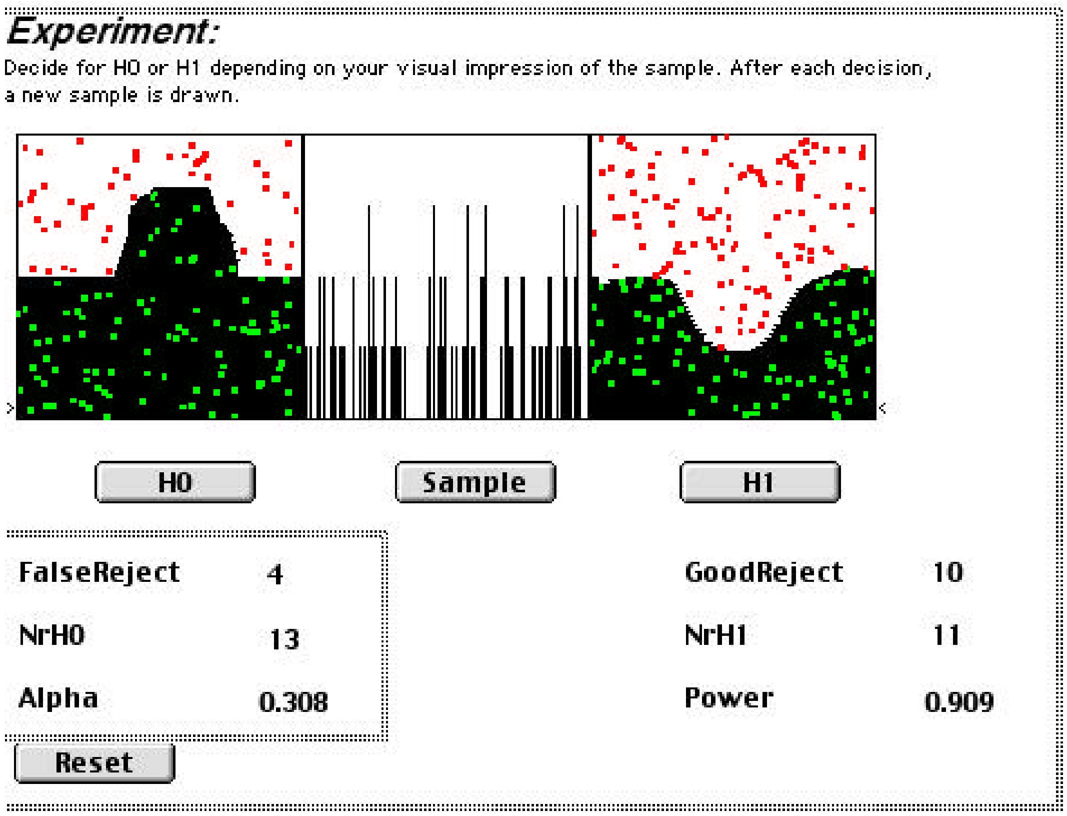
3. Response Groups

Example 2, taken from a beginner's course in statistics, is used to introduce the concepts of error of first and second kind. Using the mouse, you can draw two densities. Samples are drawn from one of these, with sample distribution shown in the middle. You have to bet where the sample came from. As you repeat the decision for repeated samples, the performance statistics are accumulated to give the level and power of this pragmatic test procedure.

Implementing this example needs more than example 1. Each of the elements (editable density, empirical density, buttons, result displays) is an independent component. The new feature required is communication between these components to share a common state. Of course you could customize components. But then you would use flexibility. One of the key ideas in component is reusability, and this asks for generic components. Message passing comes in.

The "Sample" button for example sends a message to trigger an (invisible) random number generator. This is a generic action - but thinking of document integration, you would want not all random number generators in your document to be triggered, for example if you have multiple copies. The convenient way for this example was to introduce abstract container elements. A message can be restricted to a container. This allows control flow to be structured by a simple editing or layout operation, while keeping elements reusable. Without containers, additional constructs on the programming level would be necessary.

To guarantee consistent state, in the implementation circulated we used model-view-controller (MVC) triads well known from Smalltalk. In most other applications however, the MVC paradigm turned out to be more complex than needed for statistical computing, and we returned to the simpler and much more efficient subscriber model discussed in (Sawitzki 1996).



Experiment:

Decide for H0 or H1 depending on your visual impression of the sample. After each decision, a new sample is drawn.

FalseReject	4	GoodReject	10
NrH0	13	NrH1	11
Alpha	0.308	Power	0.909

Figure 2. Response Groups. Message passing is restricted to (editable) containers. Two of these are marked by grey outlines.

4. Integrated Documents

In general, document information falls into three categories: framework information, data, and methods. Each of these needs to be pertained if a document is exchanged or edited. Framework information usually is implicit and defined by the choice of the editing environment or documentation reader. Data information generally refers to user or sample data. Methods information is the information we, the developers, provide; we need to make it accessible in a consistent and reliable way.

Proper data information is most easy to discuss. In a document model, data should be stored with a document. In a distributed environment, this may be by reference, provided authentic data recovery is guaranteed. Special considerations are required if the environment is not guaranteed. Data information is highly flexible. Interfacing to various data sources and guaranteeing reliable links to a consistent data source is of prime importance.

Framework information has most open choices, and usually it is kept implicit. In discussing statistical programming environments however, we need to make this information explicit. Which "run time" environment do we assume? And since the area is under development, we should attach a time tag: what is the life time we assume for this environment? The extreme choices are to provide the framework information explicitly, or to insist on available framework information on the user side. The examples give above are taken from a "black box" implementation. All of the information, including the framework itself, comes bundled with the document. Internally, the framework consist of a base framework. This is a host specific part. In our examples, it is a run time environment for Oberon, consisting of less than 200K. But it is not application specific - so it can be shared among various examples like others from the Voyager family, or applications from other sites, like for example BUGS, using the same framework. The base framework is augmented by various components.

For teaching purposes or document dissemination, bundling the framework is convenient (as long as no royalties are required). For more general purposes, editing or viewing habits of the user should be honored. Our preference is to use the Oberon operating system itself, but of course this can not be expected from all user. Fortunately, it is possible to implement the Oberon core as a dynamic library (DLL) (Zeller, 1999) or as a browser plug-in (Kistler and Franz, 1999). This allows using Oberon as ActiveX elements in common editors such as Word, or usual browsers such as Netscape.

While we can assume that framework conditions are met and adequate plug ins or components are installed at the user side, methods are often specialized and undergo improvements. So we cannot assume method information to be installed at the user site. If we suppose an accessible working network connection, method information can be accessed the very old fashioned way (using a client server model and invocating the method on a remote site) or by downloading method byte code and executing it locally. Java is a model doing this. For an Oberon system, a much more efficient way than the Java method is downloading the compressed code tree and executing it locally (Kistler and Franz, 1999).

For a document model, of course we would like to include method information together with our data in a document to keep the information self contained. However we cannot assume that the run time environment is the same as our production environment (or some intermediate editing environment). So the method information must be stored in a portable way. The black box implementation (shown above) only uses references to the methods, assuming that the information to execute these method is available at the run time environment. Imbedding a reference in an HTML page “by hand” has a similar effect - of course leaving out the support for editing. Using the Oberon environment, it is possible to go a step further. The run time information necessary to execute a component can be encoded in a hardware independent way, again using a compressed code tree as intermediate representation (Franz and Kistler 1997). This allows inclusion of executable content in compound documents, leading to a complete integration in a document model.

REFERENCES

Kistler, T. and Franz, M. (1999). A Tree-Based Alternative to Java Byte-Codes. International Journal of Parallel Programming, 27(1), 21–34.

Franz, M. and Kistler, T. (1997) Slim Binaries. Communications of the ACM, 40(12), 87–94.

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/>.

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

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

Résumé

En route des vues incluses par l'intermédiaire des groupes de réponse aux documents intégrés, nous discutons l'utilisation des composants de logiciel et documents integrées pour la programmation statistique. Des exemples sont pris du système Voyager, mis en application sous Oberon.