The Software JMulTi: Concept, Development, and Application in VAR Analysis.

With a detailed discussion of bootstrap confidence intervals for impulse responses.

D I S S E R T A T I O N

zur Erlangung des akademischen Grades
doctor rerum politicarum
(Doktor der Wirtschaftswissenschaft)
im Fach Ökonometrie

eingereicht an der
Wirtschaftswissenschaftliche Fakultät
der Humboldt-Universität zu Berlin

von

Herrn Dipl.-Kfm. Alexander Benkwitz

geboren am 27.01.1971 in Neu Delhi

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jürgen Mlynek

Dekan der Wirtschaftswissenschaftliche Fakultät:
Prof. Michael C. Burda, Ph.D.

Gutachter:

1. Prof. Dr. Helmut Lütkepohl
2. Prof. Harald Uhlig, Ph.D.

eingereicht am: 30. April 2002
Tag des Kolloquiums: 3. Juli 2002

Abstract

The thesis develops and examines tools for the analysis of dynamic multi–equation models (VAR models). First, a general concept for the integration of statistic procedures into a menu controlled software is developed. The resulting Java–library consists of configurable graphical user interface components and functions, which allow communication to the statistic software package GAUSS. This library is the basis for the software JMulTi, a menu-driven program for analyzing univariate and multivariate time series.

The use of JMulTi for analyzing VAR models is documented next. Unrestricted and restricted VAR models for the monetary sector of Germany are estimated and different bootstrap confidence intervals for impulse responses are computed and compared. These intervals are subject of a concluding and detailed analysis. It is examined whether the bootstrap methods used in JMulTi (and further suggestions, e.g. the subsampling) are able to overcome the possible inconsistency of the standard asymptotic method when computing confidence intervals for impulse responses. A Monte–Carlo–study illustrates the performance of the examined methods.

Keywords:

VAR Analysis, Bootstrap, JMulTi, Java

Zusammenfassung

Die Dissertation entwickelt und untersucht Methoden für die Analyse dynamischer Mehrgleichungsmodelle (VAR Modelle). Zuerst wird ein allgemeines Konzept für die Einbindung statistischer Prozeduren in eine menügesteuerte Software entwickelt. Die resultierende Java–Bibliothek besteht aus konfigurierbaren Oberflächenkomponenten und Funktionen, die die Kommunikation zum statistischen Softwarepaket GAUSS ermöglichen. Diese Bibliothek ist die Grundlage für die Software JMulTi, einem menügeführten Programm zur Analyse univariater und multivariater Zeitreihen.

Der Einsatz von JMulTi bei der Analyse von VAR Modellen wird anschließend dokumentiert. Dazu werden für den monetären Sektor in Deutschland unrestringierte und restringierte VAR Modelle geschätzt und unterschiedliche Bootstrapkonfidenzintervallen für Impulsantworten berechnet und verglichen. Diese Intervalle sind Gegenstand einer abschließenden und detaillierten Analyse. Es wird untersucht, ob die in JMulTi verwendeten Bootstrapverfahren (und weitergehende Vorschläge wie z.B. das *Subsampling*) in der Lage sind, die mögliche Inkonsistenz des standardasymptotischen Verfahrens bei der Berechnung von Konfidenzintervallen für Impulsantworten zu überwinden. Eine Monte–Carlo–Studie illustriert die Leistungsfähigkeit der untersuchten Methoden.

Schlagwörter:

VAR Analyse, Bootstrap, JMulTi, Java

Acknowledgments

The research was carried out within the Sonderforschungsbereich 373 (SFB 373) at Humboldt University Berlin. The author is grateful for financial support by the Deutsche Forschungsgemeinschaft. The author also enjoyed his stay at the PhD program "Graduiertenkolleg Angewandte Mikroökonomik" of Humboldt University and Free University Berlin in the very early stage of his work.

I would like to thank Prof. Dr. Helmut Lütkepohl for supervising my thesis. His valuable comments and suggestions were an important source for improving my work and for the progress of the project JMulTi.

Also, I would like to thank the members of the SFB 373 and their speaker Prof. Dr. Wolfgang Härdle for the productive and frank working environment. Furthermore, I would like to thank Prof. Michael H. Neumann for giving me an introduction to the mathematical view of the bootstrap methodology. I also enjoyed many discussions with Dr. Holger Bartel about this subject.

The work on a software that could execute procedures of statistical software packages was driven by the simple idea of having at least two instead of one statistical programs running on the computer. When it became clear that this idea could be realized, more effort was put into that project and Markus Krätzig joint the development. He also contributed the final name: JMulTi. The graphical user interface of the current version was written by the Alexander Benkwitz and Markus Krätzig. The econometrics of JMulTi and

the thereby used GAUSS procedures had been the work of many people (in alphabetical order): Alexander Benkwitz, Ralf Brüggemann, Markus Krätzig, Prof. Lütkepohl, Prof. Teräsvirta, and Prof. Tschernig. In addition, many colleagues and students who commented on earlier versions of JMulTi helped to improve this software a lot.

Finally, I am indebted to the people and companies who provided software resources free of charge in the world wide web: Figures 2.1 and 2.12 were created using the free *Tkpaint 1.6* from Samy Zafrany (http://www.netanya.ac.il/~samy/tkpaint.html). The screen shots were made with the 20 days free trial version of *ImageForge* by Cursor Arts (http://www.cursorarts.com/). Image format conversion was done with *ImageForge* and the conversion tool *jpeg2ps.exe* from Thomas Merz (i.e. search for "jpeg2ps" at http://www.dante.de/). And last but not least, JMulTi was created by using the free program *Visual Age for Java, Entry Edition* from IBM (http://www7.software.ibm.com/vad.nsf).

Contents

List of Figures

List of Tables

Abbreviations

DLL Dynamic Link Library. A library of executable functions or data that can be used by a Windows program.

EGLS Estimated (or feasible) generalized least squares.

IDE Integrated Development Environment. Programming environment with many tools for default programming tasks.

IPC Interprocess Communication. Data exchange between computer programs.

GUI Graphical user interface.

OLS Ordinary least squares.

VAJ Visual Age for Java[1]. An integrated development environment for programming in Java.

VAR Vector autoregression.

VCE Visual Composition Editor. A tool in Visual Age for Java that can be used for visual programming.

VEC Vector error correction.

[1] http://www7.software.ibm.com/vad.nsf

Chapter 1

Introduction

Many economic time series are analyzed with vector autoregressive (VAR) models. This framework was set up by Sims (1980) and since it had been deeply explored. There is a considerable set of theoretical results. Some of them are available in many commercial and non-commercial software packages. This makes it easy to apply the VAR framework in empirical research. Nevertheless, there remain open questions.

An econometric problem is related to the inference in the impulse response analysis. It is known that the first order asymptotic distribution can be degenerated in some cases (Lütkepohl (1993a, p. 100, Remark 1)). Even if the asymptotic distribution is well behaved, confidence intervals based on this distribution can be unreliable (Griffiths and Lütkepohl (1989)). It is therefore interesting to investigate alternative methods for computing confidence intervals. This is done in Chapter 4. This chapter analyzes the question whether bootstrap confidence intervals can overcome the known problem.

In addition to that question, some authors (e.g. Runkle (1987)) raised critics about the size of confidence intervals. They argue that the intervals are so large that an interpretation of significant results is possible only in a few cases. An often proposed solution is to include more information in model building.

Chapter 3 illustrates this approach on a small German money demand model.

The discussion on the inferential procedures in Chapter 4 ends with a clear recommendation to use a specific methodology in empirical research. In addition to that, the work on the methods in this chapter produced many software algorithms that can be used in empirical research right away. This lead to the question in which way the software can be provided to empirical research. Since this is a question of general interest some effort was made to find an applicable answer.

Empirical research can most conveniently apply econometric methods in software with a graphical user interface (GUI). A GUI frees the analyst from knowing the software that was used to program the numerical implementation. Many well known software packages for the VAR analysis have a GUI. However, they might limit the ability to apply new methods. It is not clear whether and when these programs adopt new results of methodological research. On the other side econometric method research produces many excellent software algorithms that lack, however, a GUI.

A possible solution would be to provide a GUI in addition to the numerical implementation of the statistical methods. This GUI queries input arguments from the user and displays the results in a convenient manner. However, there does not exist a closed framework for connecting statistical procedures with GUIs. Such a framework can ease the development of GUIs since they do have special needs like input of data, display of results, matrix editing etc.

Chapter 2 proposes a framework for creating GUIs of statistical procedures. The key idea of this framework is to connect a graphical front end with statistical procedures. Reusable software components reflecting the special needs of "statistical" GUIs are developed. They are assigned to standard tasks which are exactly identified.

Statistical methods represented by GUIs can be connected in a menu-driven

manner. This allows to use the results of method calls in an early stage of the analysis as input arguments for method calls later. It has turned out that complex programs, for example JMulTi, can be created with this approach. These programs are easy to maintain and to extend because of their component oriented design. The development of the software JMulTi was started by the author in order to make bootstrap inference in the impulse response analysis easily available to the VAR–modeller. The underlying VAR model is quite general. In addition to the autoregressive part, it includes an arbitrary number of deterministic variables and exogenous variables, the latter optionally with a lag structure. Subset models can be specified, either manually or by using a search algorithm. The analysis of cointegrated systems is also possible.

Chapter 3 discusses the general outline of JMulTi and documents parts of the VAR analysis. More methods and model classes were added in the past to JMulTi, for example, tools for the nonparametric time series analysis. The econometrics of today's JMulTi is the work of many people. Although the program has been developed in the spirit of MulTi1.0 (Haase et al. (1992)), the differences are so great, that it is not a real successor of MulTi1.0.

Chapter 2

A graphical user interface for statistical procedures

Statistical software packages are important when carrying out statistical analyses. In economics, empirical research relies on statistical software with the most common econometric methods in use. But also econometric method research utilizes these programs. Usually an econometric method is theoretically developed or adopted first. If the method is promising a numerical implementation is needed which is of course not available for new methods. This implementation is efficiently done with some specialized statistical software. It is then used in empirical or method research (e.g. to investigate small sample properties of inferential procedures).

Econometric software packages are either menu driven (e.g. PcFiml (Doornik and Hendry (1997))), command driven (e.g. GAUSS[1]), or a mixture of both (e.g. EViews[2]). This design determines how user and software can interact: Either by selecting command choices from various menus displayed on the computer screen or by typing the commands directly into a textual input area.

[1] http://www.aptech.com

[2] http://www.eviews.com

As Yamamoto et al. (2001) pointed out the menu structure and the menus themself apparently decide to a great extend about the usefulness of the statistical software although the total value is affected by many other factors. Menu structures suffer the drawback that they are in general not extendible. The menus can be seen as a closed system. On the other hand command driven software is easily extendible but lacks in general a sufficient user guidance. However, there are already approaches in order to overcome this last drawback. For instance, XploRe[3] and Ox[4] provide commands that make statistical procedures interactive.

This chapter develops an approach for creating a graphical user interface (GUI) for an existing command driven (statistical) software. A GUI is a graphical (rather than purely textual) program interface that takes advantage of the computer's graphics capabilities to make the program easier to use. Well-designed graphical user interfaces can free the user from learning complex command languages. An intermediate step in the development of user interfaces between the command line interface and the GUI was the non-graphical menu-based interface, which allows interaction by using a mouse or function keys rather than by having to type in keyboard commands. For instance, MulTi 1.0 Haase et al. (1992), Lütkepohl (1993b) is based on such a non-graphical menu-based interface. Now GUIs have matured and become the norm. In most situations they are now firmly established as the preferred user interface for end users in most situations when doing the statistical analysis in empirical research.

The general principle of the presented approach is to collect and generate all input data by the GUI application, and let the user select statistical analysis tools which are translated into procedure or function calls to the statistical command driven software by the GUI program. The GUI application also

[3] http://www.xplore-stat.de

[4] http://www.oxmetrics.net

receives the results and can display them in any way wanted by the user. This means that the analyst does not need to know how to use the underlying statistical software package, nor how to apply procedures correctly provided by other people. This is certainly an advantage when dealing with purely command driven software. Menu driven programs can be used right away and make it unnecessary to learn the special command syntax or to study procedure documentation in detail.

The concept strictly separates GUI application (user guidance) and statistical computation. This allows to provide numerical implementations of statistical methods independently of user guidance. Another approach is to mix user guidance and statistical computation. This is for instance currently provided by XploRe and Ox. However, in terms of applicability and re-usability of the statistical procedure the idea of strict separation is better. This approach was used before by, for instance, Liu et al. (1995).

Special emphasis is put on the idea to have a technique easily to apply but still flexible for creating a graphical front-end or GUI for statistical procedures. It is assumed that the GUI is created **after** having implemented a statistical method numerically. This means that a second programming process must be started when the (statistical) programming is done. This chapter also discusses strategies how this second programming can be achieved with low effort. This approach does not primarily deal with the question how the GUI application interacts with the statistical software, i.e. whether a stand-alone or network solution is used. This work provides a stand-alone solution (i.e. the GUI and statistical programs run on the same computer), see Subsection 2.2.4 but it can also be extended to a network solution.

The first section in this chapter will discuss the choice of programming language. Analysis, design, and implementation of the task "Creating a GUI for statistical procedures" are carried out in the second section. Building complex

applications is treated in the third section. Special emphasis is put on the question of how to model the program flow or program logic. The forth section deals with efficient GUI-programming. Finally a simple real life example is discussed in the last section with detailed treatment of visual programming.

2.1 Preliminary considerations

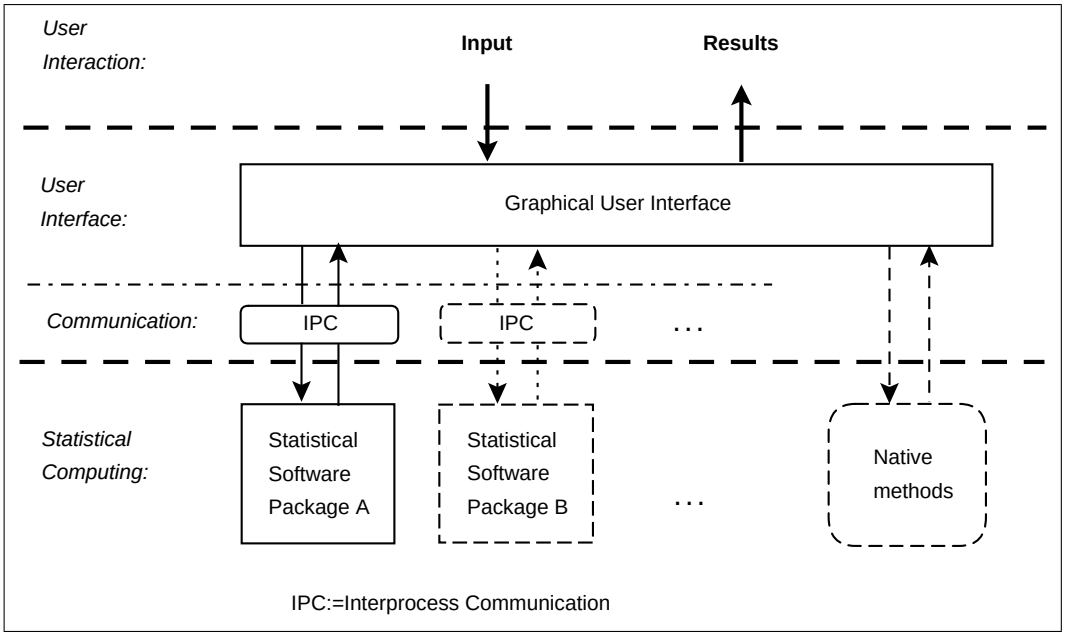


Figure 2.1: Software architecture of the concept

The architecture of the concept is shown in Figure 2.1. The user interacts with the graphical user interface which in turn communicates with a statistical software package via some so called interprocess communication (IPC). IPC is the exchange of data between two or more processes or applications. This exchange takes place solely through software, without any human intervention. As it can be seen in the figure the approach can utilize more than one statistical software package. Furthermore, statistical computing can also be executed by means of so called native methods (lower right box in Figure 2.1). A native

method (e.g. *C*-code) is program code that after compilation, runs under a specific operation system. One could think of having a numerical implementation of an estimator in GAUSS and in *C*. The advantage of native code is primarily performance. For instance, people frequently write dynamic linked libraries (DLLs) for GAUSS to speed up computation (Cribari-Neto (1999)). Furthermore the communication level is not needed (see Figure 2.1). This chapter deals primarily with the question of how to model the two layers between the bold dashed lines in Figure 2.1, the user interface and communication layers.

2.1.1 Choice of programming language

The choice of the programming language (and thus indirectly the programming technique) is determined by the task to solve. As pointed out, a GUI-application is needed that uses statistical software packages for doing statistical computation. A convenient way to solve this task is to use the object oriented programming approach and visual programming (see Section 2.5).

Object oriented programming is a programming technique in which programmers define types of operations (functions or procedures) that can be applied to different types of data (data structures or objects). In this way, the data structure becomes an object that includes both data and functions. In addition, programmers can create relationships between one object and another.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules independently of other modules. They do not need to be changed when a new type of object is added.

An object-oriented design can be kept simple and is easy to understand. Once designed, individual objects can be implemented and tested separately. Once finished, each object tends to be robust and bug free. As changes to

the system are made, existing objects continue to work. And as existing objects get improved, their interface to the outside world stays the same, so the whole system continues to work. It is this ease of change and robustness that really makes object oriented development different, and well worth the effort Wampler (2001).

To perform object-oriented programming an object-oriented programming language (e.g. Java, C++ or Smalltalk) is needed. The language Java was chosen since it is viewed as being superior in terms of some characteristics that are discussed below. But before some technical terms are explained.

Technical terms

Some of the frequently used terms which are related to object orientation and Java are explained here. A comprehensive treatment can be found in any introductory book on Java or in some of the Java online tutorials available on the world wide web. Some definitions are taken from Sun (2001).

An *application* is a program that can be executed directly from the operating system.

An *applet* is a program designed to be executed from within another application. Applets cannot be executed directly from the operating system. Web browsers, which are often equipped with Java plug-ins, can interpret applets from web servers.

A *class* is a template for creating objects. A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind. The process of creating objects is sometimes referred as to creating an instance of a class.

An *object* is an instance of a class. An object shares the behavior of all objects of the same class (defined by its methods), but each object can have a different state (represented by their variables).

An *interface* is a contract in the form of a collection of method and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface.
A *JavaBean* (or short *bean*) is a class that can be visually composed into composite beans, applets, and applications using visual application builder tools (see Section 2.5).
Run-time is the time at which a program executes. This is contrasted with the *design-time*, the time during which program code is written.

Advantages of Java

The most important advantages in terms of solving the programming task are discussed next. For a more detailed treatment of specific Java features see, for instance, Gosling and McGilton (2002).

Java is easy to apply since it has relatively few commands. Although Java has pointers (called references), it enjoys high pointer safety by automatic dereferencing, and lacking pointer arithmetic. Typical pointer errors as in C++ are not possible. Furthermore, Java has an automatic garbage collection. This means that the programmer does not need to deal with error prone memory management (memory leaks).

Java is platform independent. The Java compiler does not generate machine code but platform independent byte code. This byte code can run on every machine where a Java interpreter (= Java Virtual Machine, short: JVM) is installed. The JVM translates the byte code in the machine dependent code. In order to run a Java program on different operating systems it is not necessary to recompile the code or rewrite parts of the code.

Another important feature of Java is multi threading. Multi threading is the perceived or actual ability of a software to do more than one task at the same time. This is a prerequisite for executing code that starts some method

call (e.g. in GAUSS) and waits for the method to finish without blocking other parts of the application.

Many people work with and on Java. Therefore, it gets permanently improved. There exists a large body of literature (e.g. Eckstein et al. (1998), Schader and Schmidt-Thieme (1996)), and a large number of detailed and free tutorials and learning guides in the world wide web (e.g. Sun (2001), Gosling and McGilton (2002))). There are many already tested and ready to use Java libraries. The broad information base and the many programmed solutions reduce the work of the programmer considerably. It is also worth noting that the Java compiler can be used without paying fees. Furthermore, Sun grants anyone the right to redistribute the Java runtime environment. Any self written Java application can be combined and shipped with a copy of the Java interpreter and finally represents a self-contained program running on any major computer platform.

Disadvantages of Java

Following the discussion of Java's disadvantages in internet news groups (e.g. comp.lang.java.*) or in publications the most often mentioned disadvantages of Java are speed and the difficulties to change from another programming language to Java (or to learn Java).

First, Java is said to be slow. This statement is imprecise because different Java interpreters (or JVMs) provide different performance. But before analyzing speed available, speed *needed* is discussed. The main task of the JVM is to create GUIs, managing them, collect input data, generate computational requests, wait for and display results. Most of the time the JVM responds to user interactions. This might include to change the GUI (e.g. redraw parts or the entire application window), prepare data for statistical computation, or ask some other software to execute the statistical computation. The JVM is *not*

used for performing extensive statistical computations. Here, Java is assigned to a task that does not need outstanding performance. In trade for the lost speed (compared to *C, C++*) important features are gained: Increased ease of porting, increased speed of getting code up and running.

As mentioned before, speed of Java applications depend on the JVM used. The concept outlined in this Chapter is based on the idea that it can run with any JVM starting at release 1.2. The recommendation is, however, to use the most recent release of the Java interpreter because of JVM speed and bug fixes. In order to illustrate the performance differences a small experiment was conducted: The time needed to perform two simple matrix operations is reported in Table 2.1. The first operation is simply to copy a (100×100) matrix. The second task is to compare two (100×100) matrices. This is done for the worst case, i.e. the compared matrices contain the same information which forces the algorithm to compare 10^4 matrix elements. In GAUSS syntax this is

```
A=rndn(100,100);
A=B;       @ first operation  @
call A==B; @ second operation @
```

The operations were translated in Java code. A deep copy of two dimensional arrays is performed here. Execution time was measured for performing both operations 1,000 times using different software:

The time GAUSS needs to perform the tasks is also reported and can be seen as a benchmark since GAUSS is a matrix oriented statistical software package. It is optimized for matrix operations. Of course, the speed of *C*-code is theoretically even faster. The table shows a significant gain in performance when using Java 1.3.1 instead of Java 1.2.2.

A widely accepted solution to performance problems is code optimization. It requires to evaluate and improve the time critical code segments. However, for the two tasks in Table 2.1 better Java code can hardly be obtained, for reasons see Budimlic et al. (1999). It is widely accepted that the performance

Table 2.1: Performance of different JVMs and GAUSS. The numbers reported are the execution time in seconds for performing the respective task 1,000 times. Computation was done under Windows NT 4.0 with a Pentium 200MMX and 128MB RAM.

Software	Copying matrices	Comparing matrices
GAUSS **3.2**.29	1.5	1.6
Java **1.3**.1	2.2	5.2
Java **1.2**.2	3.7	6.4

of the current Java technology is not sufficient for scientific computation. But performance is sufficient for GUI applications (e.g. run JMulTi) because they are less computer intensive.

The second disadvantage is the use of another programming language (in this case: Java). However, using advanced programming tools and techniques helps to overcome this disadvantage. Section 2.5 discusses so called integrated development environments and visual programming. These tools make the task of writing the (Java-) GUI application manageable for people having some general programming experience but lack the specific experience with Java. They were also used to create the software JMulTi introduced in Chapter 3.

Spin-offs of using Java

It may be worth noting that Java has a number of intrinsic advantages that have not been discussed yet but are nevertheless worth mentioning. The following features are consequences of using Java and can be used by a Java programmer or can be provided by programming tools with almost none or

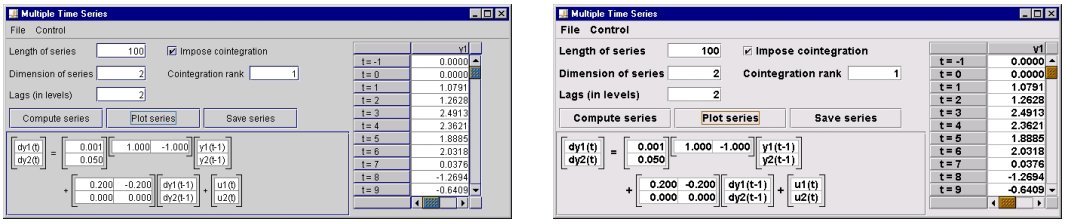


Figure 2.2: Example showing the same application in desktop mode (left) and presentation mode (right).

little effort. Notable points are:

- Pluggable "look and feel".

  This means that different visual modes of the GUI are possible. One can make the GUI look like a Windows, a Macintosh, a Motif[5], or a Java application. It is also possible to switch between a contrast rich, oversized *presentation mode* used in lectures and an accustomed colored, normal sized *desktop mode* used in everyday work on the PC. Figure 2.2 shows two screen shots demonstrating these two modes. Note that in both cases the application windows have the same geometrical size.

- Running as an applet in an internet browser.

  With respect to the security constraints, any Java application is ready to run as an applet. For example, Figure 2.3 shows the same program running as an applet in a browser window and as a stand-alone application. The communication between the applet and the statistical software (the IPC) must take into account this usage, e.g. by modeling a network communication which is of course slower than the IPC introduced in Section 2.2.4.

[5]Motif is a set of user interface guidelines created by the Open Software Foundation (http://www.opengroup.org) that specify how an application should look and feel (or react on user interaction). It is used on more than 200 hardware and software platforms and has become the standard GUI for UNIX.

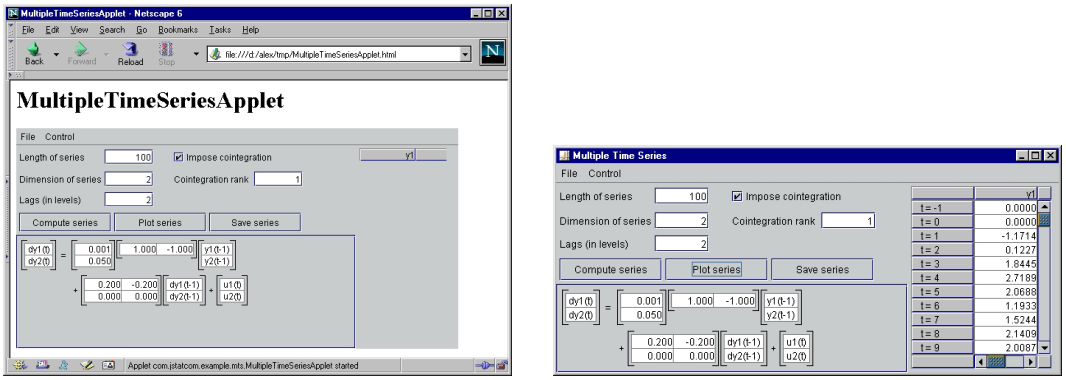


Figure 2.3: Example program running as an applet in a web browser (left) and as a stand alone application (right).

- Easy extensibility with predefined Java libraries.

  See Sun (2002) for a list of all currently available Java products and libraries. For example, a context sensitive help system can be realized with the *JavaHelp* framework. Heiss (1999) explains how help tools are developed and integrated in applications. A context sensitive help can provide information on program handling (e.g. How to select variables jointly?) and can explain statistical methods used (e.g. What is an AIC criterion?).

2.1.2 Choice of statistical software package

A statistical software package is a software that facilitates efficient coding and computing of statistical or econometric methods. This is achieved by a set of common algebraic and statistical functions and by matrix orientation.

GAUSS was chosen as the statistical computing software because of the authors experience with this software, its speed and reliability. This choice also allows to make use of the graphic capabilities of GAUSS. Therefore, graphical output is not treated here since it is provided by the statistical software.

The design of the software architecture (Figure 2.1) allows to connect the

GUI application with any statistical software package if some kind of IPC between the two programs running can be established. GAUSS has no built in communication capabilities, therefore they must be created (see Section 2.2.4 for a communication model).

An interesting feature is offered by some other statistical programs: Code written in the specific package language can be converted into *C*-code. For instance Maple, Matlab, and Mathematica provide this functionality. Compiling the *C*-code to a dynamic linked library (DLL) for Windows or shared object for UNIX allows to generate so called native methods. As seen in Figure 2.1 the native methods can be called directly by the GUI application without the communication layer and without the statistical program. This is certainly an advantage but also has the drawback of reduced functionality in terms of debugging.

2.2 Concept specification

This section describes the software concept in detail. In the beginning the matter of this chapter *"Creating a GUI for statistical procedures"* is developed theoretically. A model for controlling a command line statistical software package is needed as well as a general model for statistical analysis. Then both models are decomposed into well sized subtasks. The subtasks must be sized in such a way that an efficient implementation in Java is possible. Finally it is shown how GAUSS can be used as a statistical engine. In the next section a GUI for statistical procedures is created by re-composing and configuring the new software parts in a meaningful manner.

Since statistical analysis is of interest here there is a need to distinguish pure statistical data from the remaining program data. Therefore, in the following the term *statistical data* is used for all data in the GUI application that are

input or output data of the statistical procedures.

2.2.1 Two basic models

A model for executing a GAUSS procedure from a GUI

The most compelling question is how a GUI application can use GAUSS as a statistical engine. When doing statistical computation in GAUSS (in general in any statistical software package) from a GUI-application three basic steps can be identified:

1. Enter input arguments in the GUI application. This can be done by reading in data files, editing parameters, making selections, and/or generating input data.

2. Select a statistical method which will execute the statistical computation in GAUSS. Any statistical procedure is executed with m input arguments and returns n results, $m, n \geq 0$. Since the input/ output data are supplied/ displayed by the GUI application, this step can be split further into:

   (a) Transfer input arguments to GAUSS.

   (b) Execute the GAUSS procedure(s) of interest.

   (c) Transfer results to the GUI application.

3. Display results in the GUI.

The three steps show the need for modeling (a) statistical data, (b) GUI-components for editing and displaying that data, and of course (c) the need for modeling the functionality of GAUSS, for example to execute a command line.

A model for statistical analysis

The rather general definition of a GUI above does not describe the requirements of a GUI for statistical analysis. Therefore, a simple model of statistical analysis is needed.

Statistical analysis usually consists of several steps: Data sets are opened, visualized, and transformed. Basic statistics are computed and pre-tests conducted. After these preliminary steps models are selected, fitted, and validated. This process is repeated until a statistical sound model is found which, eventually, is interpreted with further tools. In order to proceed, the analyst must make decisions in each analysis step. These decisions are supported by statistical methods.

In empirical analysis it is often the case that model fits, tests etc. are applied on alternative data. The sample size is varied, variables are excluded or included, method parameters are changed. It turns out that the same set of methods is applied on changing parameterizations. For better comparison and judgment this is done simultaneously.

This approach leads to the requirement to have templates or masks for the data analysis. These masks must be linked with known statistical procedures from the method set. Real life objects are created and filled with statistical data during the analysis. This guarantees the ability to apply the same methods on different data and to compare results easily.

2.2.2 Sub tasks derived from basic models

Statistical data:

A model representing GAUSS data types Aptech (1996) is needed. This model must include at least the features name and data. Furthermore, it must provide methods for reading and writing data and may provide methods for simple data manipulation.

Accessibility of statistical data:

This means the ability to reference objects representing statistical data from any part of the application. The idea is to administer statistical data centrally and access it simply by its identifying name. With this mechanism statistical computation can easily use data from and provide data to other parts of the application. It also is a convenient way to configure visual software components for editing and displaying statistical data. At design time only the identifying name of the statistical data must be configured. The 'correct' data is found automatically at run time.

Visualizing statistical data:

This is one of the core tasks. The software must be able to show the user the state of the statistical data (display results) and to allow the user to edit them (enter inputs). Common display and edit mechanisms use textual input areas. However, Java's *Swing* packages provide a rich set of more elaborate software components that can be directly linked to statistical data.

Input validation:

The user can make mistakes when editing statistical data. The error can be a simple typing error or an implausible input (e.g. choosing a negative lag length). Without validating the data the statistical procedure may simply return with an error message or, worse, with erroneous results that are viewed as correct results. A well configured validation mechanism prevents the editor component to hand over nonsense data to the statistical data model. As a consequence the statistical analysis does not proceed as long as wrong input data is being edited. Implausible input is intercepted at the point where it is made which makes the GUI-application more comfortable to apply.

Access to the software GAUSS*:*

The steps for executing GAUSS code from Java must be modeled (e.g. executing a command line) as well as some communication with GAUSS since this is not

provided by GAUSS. Furthermore, an administration (possibly queueing) of all procedure call requests is necessary. Since the user does not have direct access to GAUSS and cannot control it directly this mechanism prevents that current statistical computation is interrupted or disturbed by new request to the statistical engine.

2.2.3 Implementation in Java

This section concentrates on the implementation of the tasks identified in the previous section. It tries to use as much as possible from predefined Java concepts and libraries. Especially it heavily depends on the Java library *Swing*[6]. Any application written with this concept is a *Swing*-application.

The resulting Java library contains 54 classes, 6 interfaces, and native methods (written in *C*) for the communication with GAUSS. The library and the native methods were successfully tested with Java 1.2 and 1.3 under Windows 95, 98, 2000, and NT4.0. In the following it is called **JStatCom** (Java graphical user interface for Statistical Computing). A complete documentation is provided in Appendix B.

Package structure

The library JStatCom is organized in packages. The structure can be seen in the Table 2.2.

Classes which implement sub tasks

Statistical data:

Statistical data is modeled by the class

`gauss.GaussData`.

[6] *Swing* was the code name of Sun's project that developed a new set of GUI components. Although it is an unofficial name, it is frequently used to refer to the new components.

Table 2.2: Package structure of the library JStatCom

Package	Content
`gauss`	Representation, visualization, and computation of statistical data; Modeling the software GAUSS.
`gauss.control`	Convenience classes for controlling and visualizing all statistical data available application-wide.
`util`	Collection of utility classes.
`util.component`	Adopted and improved *Swing* components.
`xlm`	GAUSS communication files

A GaussData object is constructed with a symbol name and optionally with some data. The symbol name is treated case insensitive and cannot be modified later. GaussData objects can represent different data types. Currently, the GAUSS data types *empty*, *matrix*, *string*, and *string array* Aptech (1996) are implemented. The data type *character matrix* is not implemented. It is also possible to handle missing values. In a GaussData object they are represented by the data type *matrix* with some or all matrix elements set to **NaN** (Not a Number). These **NaN**s are correctly communicated to GAUSS and recoded appropriately.

The GaussData class defines methods for accessing and manipulating the data. In order to synchronize the read/ write access by different threads, all data manipulation is done in the so called event dispatching thread[7]. It was chosen here for convenience and because this class serves as the data model for many GUI components which are displayed and updated in that thread. For

[7]The event dispatching thread should execute all code that might affect or depend on the state of a GUI component. This rule comes from the synchronization requirements of any multi threaded application. However, there are exceptions to that.

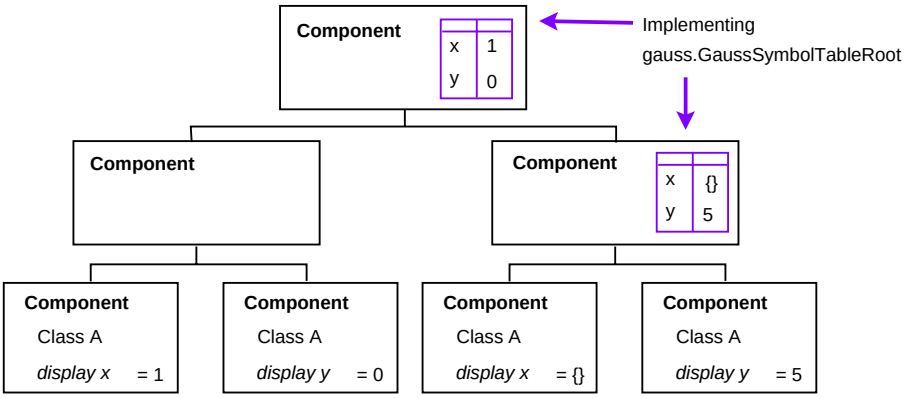


Figure 2.4: Component hierarchy and data accessibility

instance, a GaussData object can be used as the data model for a

`gauss.GaussDataTable`

which is a software component for displaying and editing statistical data.

Any GaussData object can notify interested listeners about its change of state. The mechanism which notifies of table model events is inherited from

`javax.swing.table.AbstractTableModel`

in principle. Originally, it notifies registered listeners every time the TableModel is edited. That mechanism is improved in such a way that notification only takes place if the editing actually changed the data content.

A class that self-evidently wants to get informed when the data model changes is a class that displays statistical data at the GUI. The notification is needed in order to update its visual appearance. But it is also possible to use that information to automate program flow. One might think of making analysis steps or analysis methods not available until a set of specified data objects (e.g. the input data) meet predefined needs (e.g. contain sensible values). One could also automate resetting of results if some input data change. All that can make the software intuitive and safe to use.

Accessibility of statistical data:

In order to access GaussData objects from any point of the Java application

(a) GaussData objects are identified by a name

(a) GaussData objects are administered by a special object

(b) A search mechanism based on the component hierarchy finds the right administration object.

For the administration the class

`gauss.GaussSymbolTable`

was developed. In order to get reference to a GaussData object from this class the method

`GaussSymbolTable.getGaussData(String symbolName)`

can be called with the identifying name as an argument. It returns reference to the GaussData object with the argument name. If a GaussData object with the specified argument name does not exist it is created. Every closed analysis step should have at least one (local) GaussSymbolTable object. (Local) GaussData objects behave in this case similar to local variables: The same identifying names can be assigned to different local GaussSymbolTables without mixing up the data.

For implementing convenient data accessibility the so called containment hierarchy of the visual software components is used. Containment hierarchy means that each visual *Swing* component, i.e. each

`javax.swing.JComponent`

has a parent component and optionally one or more child components. Therefore, even the simplest *Swing* program has multiple levels in its containment hierarchy. It is easy to define a component in this hierarchy that holds a GaussSymbolTable and thus acts as the root for providing statistical data. The root can be any JComponent implementing the interface

`gauss.GaussSymbolTableRoot.`

When the application is created at run time, all single components are added to their parents starting at the topmost component. If a reference to a GaussData object is needed the program code can 'walk up' the hierarchy until a GaussSymbolTableRoot is found from which it can reference any GaussData object. Figure 2.4 illustrates this idea. Finding an appropriate GaussSymbolTable is conveniently implemented with the static method

```
gauss.GaussObjectLinker.
    findGaussSymbolTableForComponent().
```

The accessibility mechanism is implemented in many other classes of JStatCom, e.g. in classes that display statistical data. It allows to configure components of these classes simply with the identifying name of the statistical data at design time. At run time the reference to the contextual right data object is found and maintained automatically. For instance in the bottom line of Figure 2.4 are two objects created from class A. They display data with the name `y`. In the analysis context represented by the left side of the tree the actual data is `y=0`, in the analysis context of the right side it is `y=5` .

Visualizing statistical data:

Three classes are defined for displaying statistical data (GaussData objects) in the application window:

Class	Displayable dimension	Editable
`gauss.GaussDataTable`	(M, N)	yes
`gauss.GaussDataTextField`	$(1, 1)$	yes
`gauss.GaussDataLabel`	$2 \times (1, 1)$	no

This list can easily be extended by other input components like check box, combo box, list, etc. As outlined before, only the name (identifier) of the statistical data must be specified at design time. The contextual right data object is referenced and displayed at run time.

If the dimension of the statistical data exceeds the available space (e.g. displaying a time series with 100 observations) the GaussDataTable should be used together with a

`gauss.GaussDataTableScrollPane.`

This specialized scroll pane can be configured to display a minimum and maximum number of rows and columns. The actual number of displayed rows and columns depends on the available space in the application window at run time. The available space might change dynamically (e.g. resizing the window). In this case the GaussDataTableScrollPane adopts to the new space automatically.

The GaussDataLabel can show up to two matrix elements of a GaussData object by using a *C*-style sprintf() format string. The elements can be specified by configuring row and column indices. The GaussDataTextField displays and edits one element of a GaussData object that is also specified by a row and column index. Whenever the underlying GaussData object changes GaussDataTables, GaussDataLabels, and GaussDataTextFields update automatically.

The editable components can be configured with information about the type and range of valid values. The range is specified with a lower and upper bound $\{a, b\}$, with an interval type ((a, b), $(a, b]$, $[a, b)$, or $[a, b]$), and with the input data type (integer, real number, or string). The input validation algorithm is triggered if a component finishes editing. It then uses the configured information about the legal values.

Input validation:

All inputs are made in a textual input field. The textual input must be translated into the underlying data model when the input is finished. Two events signal the end of an input: striking the Return-key and moving the input focus permanently away from the editing component (e.g. a mouse click on a menu).

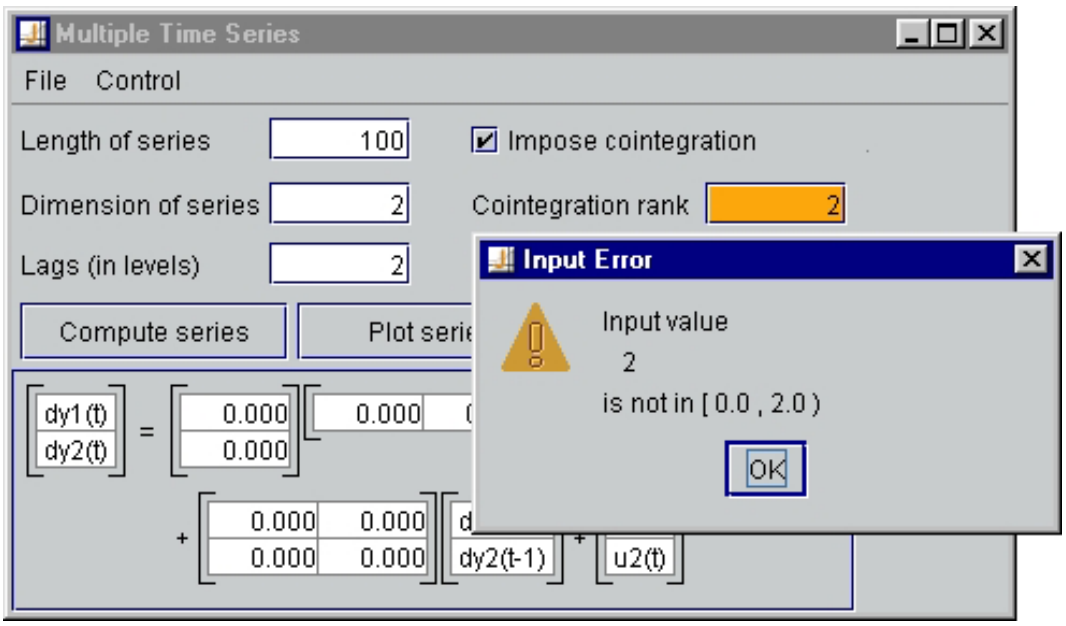


Figure 2.5: Error message triggered by the input validation algorithm. The error dialog is modal, i.e. the remaining part of the program is blocked while the dialog is shown.

But before translating and writing the input to the underlying GaussData object the input validation mechanism starts.

There are at least three different ways text input can be validated: key-stroke level, focus level, and data model level. They all turned out to be insufficient. Therefore a modal validation mechanism was developed. It is implemented in the class

util.component.InputValidatingTextField.

It does not release input focus while the editor contains invalid text input. This means that the user cannot proceed the analysis as long as wrong input data are in the text area of the editing component. In order to cancel editing and to prevent this input field from locking (e.g. when the valid range is empty), striking the ESC-key restores the old value and unsets the focus from this component. This class is extended by

`util.component.ModalTextField`

which is extended by

`gauss.GaussDataTextField.`

Figure 2.5 shows an error dialog as an example. It informs the user that the cointegration rank r has to be $0 \leq r < 2$.

Access to the software GAUSS*:*

The class

`gauss.Gauss`

holds the connection with a running instance of the software GAUSS. It contains methods for starting and terminating this program, for writing and reading data, and for executing command lines, loading libraries and DLLs in GAUSS. Furthermore, a registration and queuing mechanism prevents the Java application from mixing up different statistical computation requests.

The Gauss object is found by "walking up" the containment hierarchy of the visual software components. Any JComponent in the hierarchy can contain a Gauss variable. It is identified by implementing the interface

`gauss.GaussEngineRoot.`

Any class implementing this interface promises to provide reference to a Gauss object. It is implemented by the class

`gauss.GaussFrame.`

Whenever a component in the containment hierarchy wants to execute some computation in GAUSS, it simply walks up the hierarchy until it finds a GaussEngineRoot, and executes the method

`gauss.GaussEngineRoot.getGauss().`

2.2.4 Run GAUSS as a statistical engine

Communication with GAUSS

A key feature of the concept is that only the GUI application interacts with GAUSS (see Figure 2.1). This means that GAUSS must be controlled automatically. Such functionality is not provided by GAUSS and was therefore developed here. The communication for the Windows platforms was developed by the author (for GAUSS 3.2) and by Markus Krätzig (for GAUSS 3.5 and higher). If GAUSS 3.2 is used as the statistical engine, it runs with a (minimized) visible application window. Note, that in this case the ability to receive keyboard input of the GAUSS application window is disabled. If GAUSS 3.5 or higher is used it starts in the so called terminal mode. In that case GAUSS runs as a command line application and appears to run in the "background". The communication model can also be used to connect any other statistical software package as a statistical engine with the Java application.

In contrast to the platform independent GUIs the communication between the Java application and the statistical engine is implemented in a platform dependent way because of performance reasons. The model consists of a set of external functions for controlling the statistical software. The control is internally achieved with a signaling mechanism for process synchronization and a data exchange mechanism. See Appendix C for details. Since it is assumed that the GUI application and GAUSS run on the same computer (i.e. a stand-alone solution) the data exchange is implemented by using shared memory. The data exchange with files on the hard disk would be too slow and inefficient. Shared memory is the fastest data exchange available but cannot be used for network computing. In that case, another mechanism for data exchange can be implemented.

The current communication is provided by the software *Gauss Control*

(GC). It provides external functions in order to achieve the following tasks with GAUSS:

1. Start program,
2. Write data from another application to GAUSS,
3. Read data from GAUSS by another application,
4. Run command lines,
5. Run program files,
6. Test for errors,
7. Stop current program execution,
8. Quit program.

The external functions are documented in Appendix C. All functions are synchronized, i.e. they are blocked as long as the tasks 2 to 7 on the GAUSS side are executing. In this software model GAUSS cannot write data actively to the Java application. Even when returning results, GAUSS is asked to return the content of a certain variable. This is advantageous when calling procedures with many return values from which only a few are needed in the GUI application. However, actively communicating data from GAUSS can also be useful, for instance to indicate computational progress. This could be an interesting feature for future development.

The communication library is self-contained and can also be used by other programs in order to use GAUSS as a statistical engine. Here it is part of JStatCom. Six native methods defined in the class Gauss use the external functions of GC.

Since the communication library is written in *C* it is not platform independent (in contrast to the remaining Java library) and must be re-implemented

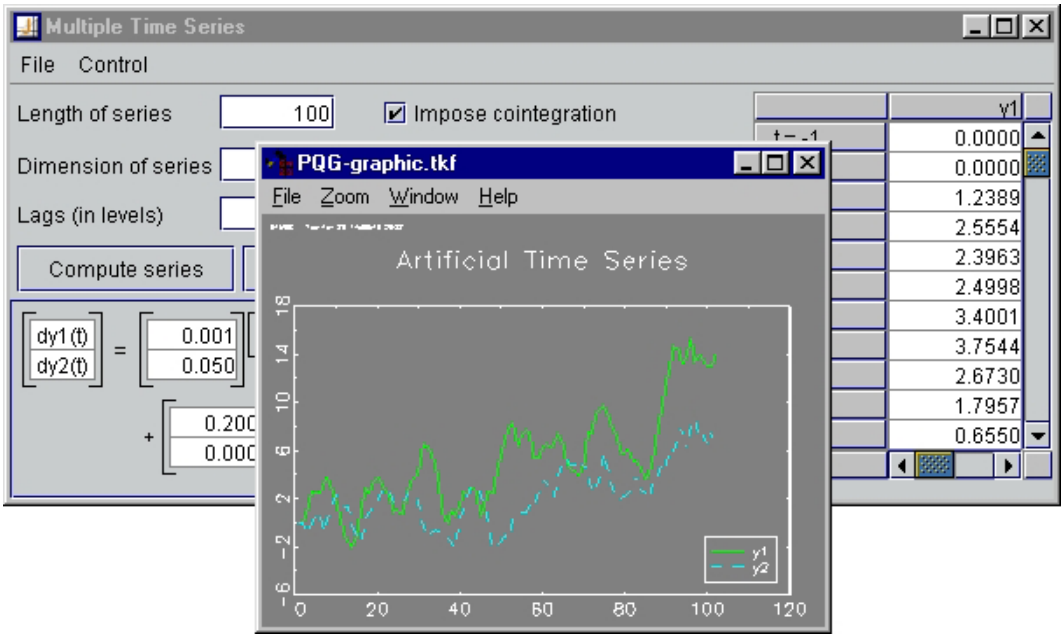


Figure 2.6: Graphics output of GAUSS is used. The picture shows a graphics window generated by GAUSS 3.2 on top of the GUI application.

for other operating systems or other communication models like network communication. A re-implementation does not affect the remaining Java library nor any Java application based on this library. The effort of creating and testing a GUI application is not wasted. It still can be used on other operating systems or in web browsers. This is an important advantage of Java ("write once, run anywhere").

Graphics output

Clear and expressive graphics are an important part of any statistical software. It has turned out that the graphic capabilities of GAUSS can be used. Figure 2.6 shows a GAUSS graphic on top of the application window. When a procedure generates some graphic output it is displayed in a GAUSS graphic window (for GAUSS 3.2, Figure 2.6), or in an external viewer (for GAUSS 3.5 and higher, because of the terminal mode). Any call to a graphics routine must be followed

by coding convention 3 of the next subsection. This ensures that the external viewer is started or the last graphics window is displayed on top of the Java application window. Chapter 3 provides elaborate examples that show the interaction of an GUI application and GAUSS for graphical output.

In general it is possible to provide graphs generated by the Java application. This provides a uniform graphical output if different statistical engines are used. However, to provide pure Java solutions to generate graphical output is beyond the scope of this work.

Coding conventions for GAUSS-procedures

In order to call GAUSS-procedures from the Java application some minor GAUSS-coding conventions must be fulfilled. In general, GAUSS procedures are used in the statistical engine exactly in the same way as in the usual Windows application. GAUSS is started as (or converted to) a statistical engine by loading the library `xlm` and the DLL `glm.dll`. They provide the communication capabilities on the GAUSS side (see Appendix C for details).

GAUSS code that runs in the statistical engine should regard the following four points:

1. Apply procedural programming techniques. Provide the procedures in a library with a unique name. Give the procedures unique names in order to avoid name clashes, e.g. `libname_procedureName`. When using this rule it is only necessary to check whether `procedureName` already exists in the library `libname` which is easier to do than cross-checking all potential libraries.

2. Try to avoid global variables. If global variables are really important they should have a unique name, e.g. `__libname_variableName`.

3. If GAUSS procedures generate graphic output place the following three

lines after the function call that displays the graphic window:

```
if __XLM_LOADED;
    dllcall showLastGraphic;
endif;
```

The global variable `__XLM_LOADED` and the function `showLastGraphic` are defined in the GAUSS library `xlm`. The above three lines ensure that the last graphic window generated by GAUSS is set in front of all other application windows on the computer screen (especially the GUI application window) and conveniently signals the user proceedings in the GAUSS procedure call. The library can still be used outside a GUI application by declaring

```
declare matrix __XLM_LOADED?=0;
```

4. If the procedure wants to report errors to the GUI application it must use the GAUSS `errorlog()` function. Arguments of this function are written to an error log file which is monitored by the GUI application.

Conventions 1 and 2 are commonly understood as good programming practice. Conventions 3 and 4 are specific to the applied software architecture but do not constrain the coding too much. Note that the code is not written exclusively for a statistical engine used by a Java application. It can also be used perfectly outside of this concept. On the other hand already existing GAUSS-libraries can be utilized by this concept.

2.2.5 Error handling

The concept and the single software components were designed and tested carefully. Nevertheless, programming errors or misspecification can still be present and can badly affect program execution. Sensible and informative error handling is important to users and programmers. It provides important information about current inabilities of the software and about possible

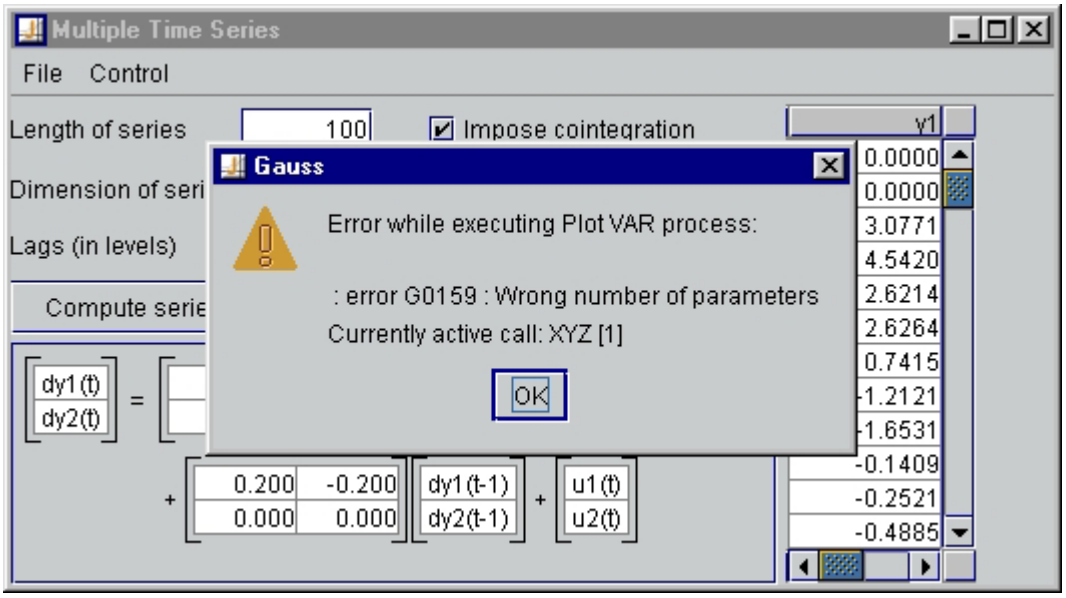


Figure 2.7: GAUSS error message reported to GUI application.

misspecified input arguments for statistical computations. At least two error sources can be identified: The Java code executed by the Java interpreter (i.e. within the self written Java classes or other Java libraries used) and the code executed by the statistical engine (i.e. self written or third party GAUSS libraries). Errors located in the Java interpreter or statistical engine are not treated here.

Java errors at run time are reported by so called exceptions. They can be caught and evaluated by the code and thus made harmless to the application. Whenever it seemed appropriate Java exceptions were caught in the self written Java classes. The ability of the GUI program to react on new user input is not affected in general. This makes the application robust. Deadlocks and program crashes of applications written with this concept were not experienced by the author.

Error handling for GAUSS computations is a built in feature of the concept. Figure 2.7 shows an example where an error message caused by wrong GAUSS code is reported. Even if the code syntax is corrected it is still possible to

formulate a computation task that results in a GAUSS error. This indicates that the requested procedure call was not finished. A run time GAUSS error typically happens when a function cannot evaluate its argument (e.g. inverting a singular matrix, using a bad matrix index). GAUSS stops executing the code and prints an error message in its textual output area. This message is read by the Java program and is reported in a message box to the user. It does not harm neither the Java program, the communication to GAUSS, nor the ability to execute another GAUSS procedure.

2.3 How to create single step GUI applications for GAUSS

This section explains how a GUI can be created from which GAUSS procedures are executed. It is called "single step" because all graphical input/ output components and control elements (buttons, check boxes, ...) can be accommodated in a single GUI. This is usually not possible in complex analyses in which a succession of different GUIs is necessary (e.g. specify model, estimate model, validate model, ...). These multiple step GUIs are discussed in the next section.

Examples for single step GUIs are provided in Section 2.6 and in the Appendix B. This section explains classes in JStatCom that were used for creating these examples.

2.3.1 The container class

All input, output, and control components are arranged on a container object. In *Swing* the container object usually is of type `javax.swing.JPanel`. A JPanel occupies a rectangular area of the computer screen. Visual components can be arranged on that area.

In JStatCom the class

`gauss.GaussPanel`

is defined. GaussPanel is a subclass of JPanel. It extends JPanel's functionality with regard to storage and query of statistical data

`GaussPanel.storeData(XXX data, String symName)`,

`GaussPanel.setGaussData(GaussData data)`,

`GaussPanel.getGaussData(String symbolName)`.

Furthermore GAUSS code can be executed conveniently from GaussPanel objects using the

`GaussPanel.gaussXXX()`

methods (see next subsection). If methods are defined that perform manipulation on statistical data or execute GAUSS code then the GaussPanel should be used as a container.

2.3.2 Execute GAUSS-code

There are two ways for executing GAUSS-code from Java.

For standard computations it is straightforward to use a combination of the methods defined in the GaussPanel class

`GaussPanel.gaussLoadLibrary(String libName)`,

`GaussPanel.gaussWrite(String symbolName)`,

`GaussPanel.gaussExec(String command)`,

`GaussPanel.gaussRead(String symbolName)`.

They load a GAUSS-library in GAUSS, write data from Java to GAUSS, execute a command in GAUSS, and read data from GAUSS by Java. In Java the data is read from and written to the next non-local GaussSymbolTable in the component hierarchy. These methods must be executed within the

`GaussPanel.gaussStart(String jobName)`

`GaussPanel.gaussRun()`

methods, e.g.

```
gaussPanel.gaussStart("Compute and plot random walk");
gaussPanel.gaussLoadLibrary("pgraph");
gaussPanel.gaussWrite("T");
gaussPanel.gaussExec("y = cumsumc(rndn(T,1))");
gaussPanel.gaussExec("xy(seqa(1,1,T),y)");
gaussPanel.gaussExec("dllcall showLastGraphic");
gaussPanel.gaussRead("y");
gaussPanel.gaussRun();
```

(see also Appendix B). The single computation steps are stacked up and passed to the *gauss.Gauss* object (see page 27) in the `gaussRun()` method. The computation is started and monitored by this *gauss.Gauss*-object when no other computations are running on GAUSS, otherwise it is queued.

It is also possible to program a procedure call by sub-classing

gauss.GaussProcedureCall

and overwrite its *runCode()* method. In this class a variable "`gauss`" is defined which is of type *gauss.Gauss*. It is set to a *gauss.Gauss* object before `runCode()` is executed. This variable must be used for defining the different computation and communication requests. For the small example above one could place the following lines in `runCode()`:

```
gauss.loadLibrary("pgraph");
gauss.writeGaussData(gaussSymbolTable.getGaussData("T"));
gauss.executeCommand("y = cumsumc(rndn(T,1))");
gauss.executeCommand("xy(seqa(1,1,T),y)");
gauss.executeCommand("dllcall showLastGraphic");
GaussData y = gauss.readGaussData("y");
gaussSymbolTable.setGaussData(y);
```

When using a GaussProcedureCall, reference to the statistical data must be defined, e.g. by handing over the GaussData objects or by providing reference to a GaussSymbolTable. The latter option was applied above by defining the variable "`gaussSymbolTable`" which was set to the correct GaussSymbolTable before executing the code. In order to execute the computation an instance

of the sub-classed GaussProcedureCall must be created and registered with a Gauss object using

```
gauss.Gauss.register(GaussProcedureCall gPC).
```

If GAUSS is busy the registration queues the call otherwise it is executed immediately.

2.4 How to create and extend multiple step GUI applications for GAUSS

The GUI application consists of several analysis steps in most cases. These steps cannot (and should not) be accommodated within a single application window physically at the same time. As a consequence the window must change its contents dynamically depending on the analysis step. This is nothing new and anybody using Windows software is familiar with that.

This section discusses classes that help to model program flow. Some classes are part of the JStatCom library, some are adopted from the *Swing* library but are also mentioned here for completeness.

It is assumed that each analysis step is already modeled or visualized with one ore more single step GUI as outlined in the previous section.

2.4.1 Available data

Sometimes it is desirable to add a new GUI to an already existing set of GUIs. For example, one might think of a new tool for performing tests on residuals that were computed in an already provided model estimation algorithm. For that, the new GUI will use statistical data computed by the other GUIs, for example the matrix of residuals. Therefore, it is necessary to know the identifying names of the statistical data objects.

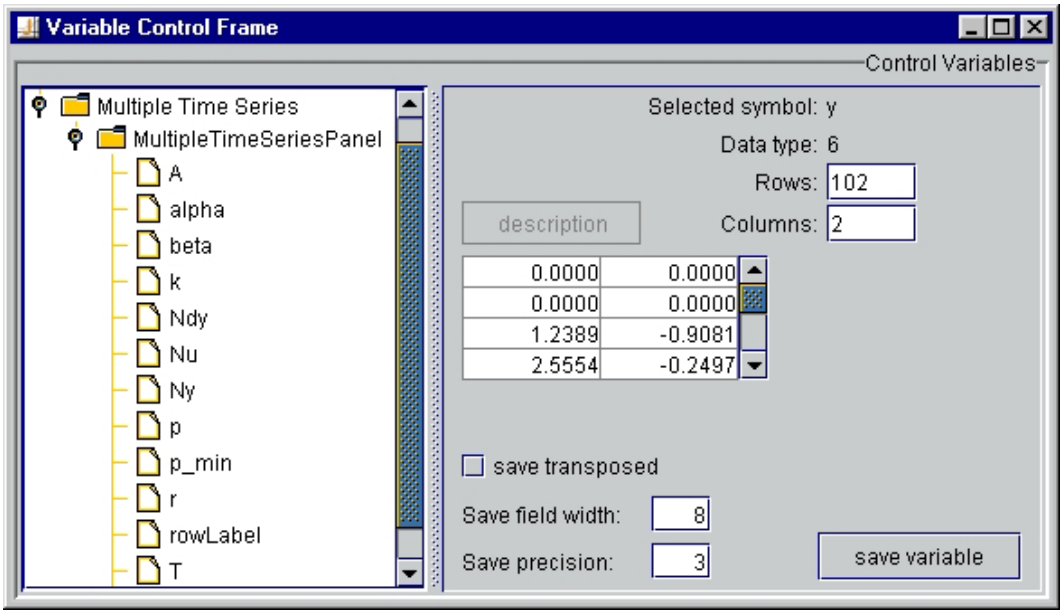


Figure 2.8: Overview of statistical data used in the program in Section 2.6.

The variable control frame can be used for inspecting all GaussData objects (i.e. statistical data) available in the application. Figure 2.8 shows the variable control frame for the example in Section 2.6.

2.4.2 Program flow

Program flow is a automatical succession of single GUIs as created in Section 2.3. It is perceived by the user when the application window changes its contents, e.g. when the program moves from model estimation to model validation. This change can be caused by the user for example by making a selection from a menu or by clicking some button.

In the Java library JStatCom two classes are provided for implementing this behavior conveniently:

`util.component.CardChangePanel` and

`util.component.CardPanelAction`.

The CardChangePanel is the parent container for all used displays (cards).

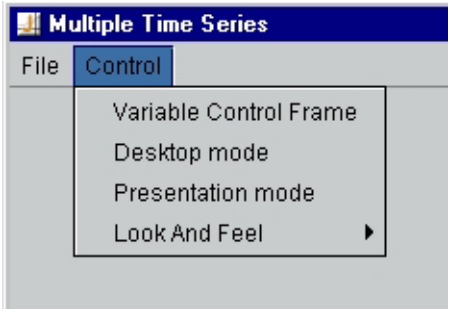


Figure 2.9: Predefined menu that can be added to any menu bar.

It must be added to the content pane, or must be set as the content pane of a JFrame or JInternalFrame. The CardChangePanel is designed to switch efficiently between the cards added.

CardPanelActions help to change conveniently between the single GUIs. Objects of this class are constructed with the class name of the GUI and with the reference to a CardSuccessionPanel where it will be displayed. Only the class name of the target GUI is used for creating a CardPanelAction because creating complete GUI objects can be time consuming. Furthermore, it is possible that a program does not need all available GUIs. At the first time the GUI object is really needed it is constructed. This spreads construction time over the time the analysis is conducted. CardPanelActions are subclasses of *Swing's* AbstractAction. It can therefore be used for creating menus and tool bars.

2.4.3 Menus as knots for program flow

Swing provides classes for creating menu bars, menus, and tool bars:

`javax.swing.JMenuBar`,

`javax.swing.JMenu`, and

`javax.swing.JToolBar`.

Tool bar buttons and menu items can be created by using AbstractActions

(e.g. CardPanelActions) and by applying the respective `add()`-method. The tool bars can be set to an application window. The menus can be added to a menu bar which is set to the application window as well.

The package `gauss.control` has a predefined menu

`gauss.control.GaussMenu`

which contains entries for displaying the variable control frame as shown in Figure 2.8 and for changing the "look and feel", see Figure 2.9. The package also contains the action class

`gauss.control.ShowControlFrame`

that can be included in any menu or tool bar. If this action is executed the above mentioned variable control frame appears.

2.4.4 Frames

The application frame is the container for all panels, menus, and tool bars. JStatCom defines

`gauss.GaussFrame`

which can be used as the top level component. The GaussFrame is a specialized JFrame that holds references to a `gauss.Gauss` object and a GaussSymbolTable. If the area within the GaussFrame is organized as a desktop (e.g. in JMulTi) it is possible to use the class

`gauss.GaussInternalFrame`

instead of the JInternalFrame class for creating internal frames on the desktop. The GaussInternalFrame is a specialized JInternalFrame implementing the interface GaussSymbolRoot. This means that any added child which queries non-local statistical data will be directed to the GaussSymbolTable of the GaussInternalFrame.

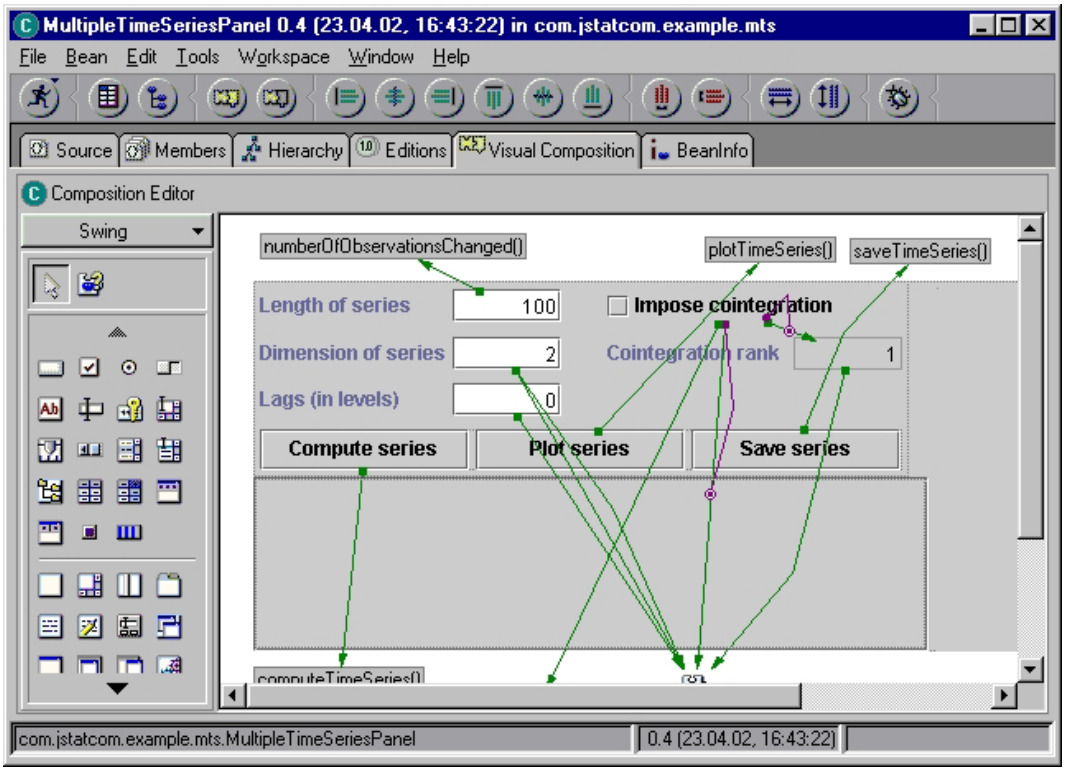


Figure 2.10: The Visual Composition Editor of IBM's Visual Age for Java with the palette of reusable beans (left) and the canvas (area with white background).

2.5 The use of an integrated development environment

The integrated development environment (IDE) was possibly invented in the 1980s when software companies like Borland delivered "visual" programming tools for programmers who needed to program visual applications. The scope of modern IDEs is not restricted to GUI programming. Complete applications can be written, tested, and finished with an IDE. IDEs are available for writing code in many programming languages. Furthermore, for a specific language there exist different IDEs in most cases.

2.5.1 Use and advantage

The classical way of programming is done in a file-based environment: Code is written in source files, compiled and linked into executable binary files. When building executable code the programmer must manage files and file dependencies. These tasks can be viewed as a standard programming exercise which is time consuming and error-prone but does not directly solve the programming problem.

An IDE performs the task of managing and compiling the code. Thus the standard programming task mentioned is delegated to the IDE. It may also contain tools for managing code versions and debugging. Using an IDE the programmer can fully concentrate on translating the problem into the program. Therefore, it is preferable to use an IDE for the programming of complex applications. An IDE may also contain tools for visual programming which is explained in the next subsection.

2.5.2 Visual programming

Programming is the translation of the problem world to the program world. The problem world might include simultaneous equations and matrix inversion, the program world deals with loops, indices, and temporary variables. The goal of computer scientists has always been to improve this match between these two worlds Green and Blackwell (1996). Visual programming is a programming approach which attempts to improve the match. It refers to software development where graphical notations and interactively manipulable software components are primarily used to define and compose programs. The goal of visual programming is to enhance the comprehensibility of programs and to simplify programming. Furthermore, visual programming should empower end users to build their own programs that otherwise would have to be written by professional programmers Schiffer (1998).

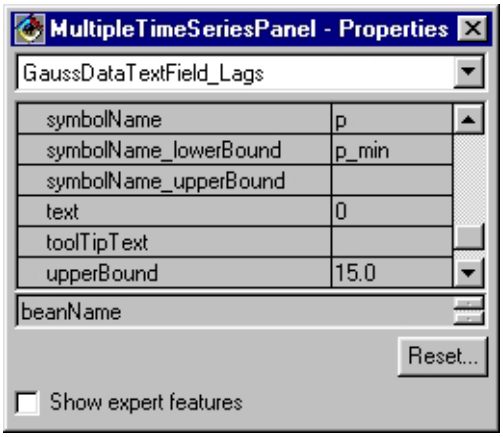


Figure 2.11: Property sheet of GaussDataTextField.

An example for visual programming is shown in Figure 2.10. It shows the Visual Composition Editor (VCE) which is a part of IBM's Java IDE, Visual Age for Java (VAJ). The VCE is used for assembling program elements visually from configurable software components called JavaBeans. To build a program with the VCE, a picture is drawn using a canvas and a palette of icons representing reusable beans. This picture specifies the set of beans that implements a function of the larger program (or bean). For beans like user interface controls, the position of controls in the picture specifies how the controls will appear in the final program. Furthermore, the VCE provides a sophisticated connection capability for specifying how components of the picture will interact to implement functions of the program. Using connections, much of the behavior of an application can be specified graphically (IBM (1999)).

Bean properties can be changed using the bean property sheet. Figure 2.11 shows the property sheet of a `GaussDataTextField` bean. For instance, the property `symbolName` is set to `p`. This means that all entered values are stored in the `GaussData` object with the name `p` if they are accepted by the verification algorithm. This algorithm tests whether the input text represents a certain data type, and is within an acceptance interval. Furthermore it can

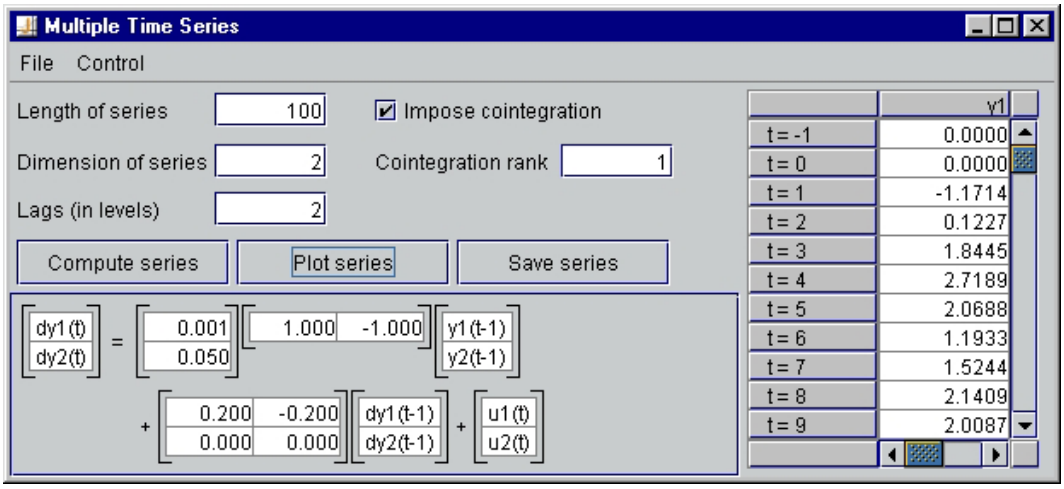


Figure 2.12: GUI of the program *Multiple Time Series*.

be seen from Figure 2.11 that the lower interval bound is set to the variable `p_min` and the upper interval bound is fixed to `15`.

The visual expressions represent appearance and function of the developed software component. They are used as graphical interfaces to the textual programming language Java in the programming environment VAJ, i.e. VAJ generates from the picture textual Java source code which is compiled and executed.

Working with the VCE does not require much knowledge of the underlying programming language Java. Typing Java code can be avoided completely for coding standard behavior. Theoretically it can be extended for specifying complex software behavior. In any case there must exist tools for configuring software elements and for visually drawing a picture of the program logic. The shown VCE and property sheet are examples for such tools. However, more customized tools must be developed in order to use all advantages of visually programming. With these tools it is potentially possible to create a complete application visually, i.e. without the need to learn Java.

2.6 The anatomy of a JStatCom-based program

This section takes apart a real program written using the approach which was outlined in the previous sections. It is called *Multiple Time Series* and demonstrates a GUI for executing two different GAUSS procedures: (a) to compute realizations of a given vector autoregressive process and (b) to plot the computed time series. Figure 2.12 shows a snapshot of the program's GUI.

Individual lines of code are not treated. Instead, it is discussed how the GUI-features of the concept are used by the application *Multiple Time Series*. This section discusses the use of specialized GUI-components, the containment hierarchy, data models, and event handling.

2.6.1 Problem description

Assume an application is needed that computes artificial time series from the VAR model

$$y_t = A_1 y_{t-1} + \ldots + A_p y_{t-p} + u_t, \tag{2.1}$$

with $t = 1, \ldots, T$ and $u_t \sim N(0, 1)$, where y_t is a K–dimensional vector of observations at time t, $A_1, \ldots, A_p$ are $(K \times K)$-dimensional parameter matrices, and u_t is a $(K \times 1)$ vector of innovations. In order to ease the specification of cointegrated processes with r cointegration relations $(0 \leq r < K)$ the VAR should optionally be specified in vector error correction form

$$\Delta y_t = \alpha\beta y_{t-1} + \Gamma_1 \Delta y_{t-1} + \ldots + \Gamma_{p-1} \Delta y_{t-p+1} + u_t, \tag{2.2}$$

where α is a $(K \times r)$, β is a $(r \times K)$-dimensional parameter matrix and $\Gamma_1, \ldots, \Gamma_{p-1}$ are parameter matrices of dimension $(K \times K)$. Equation (2.2) can be rewritten in the form (2.1) (see Section 3.2.3). Given K, p, T, $(y_{-p+1}, \ldots, y_0)$, and $A_1, \ldots, A_p$, realizations of y_i, $i = 1, \ldots, T$, can be computed recursively.

Table 2.3: Input and output variables of *Multiple Time Series*. The acceptance range of the input values is given in the last column with I denoting integers and $\mathbf{R}$ real numbers.

Description	Variable	Dimension	Legal values of matrix elements
Input			
Process dimension	K	(1×1)	I, $[1, 10]$
Number of observations	T	(1×1)	I, $[1, 5000]$
Number of lags	p	(1×1)	I, $[0, 15]$
Co-integration rank	r	(1×1)	I, $[0, K)$
Loading coefficients	α	$(K \times r)$	$\mathbf{R}$, $[-100, 100]$
Co-integration vector	β	$(r \times K)$	$\mathbf{R}$, $[-100, 100]$
Coefficient matrix for VAR	A	$(K \times Kp)$	$\mathbf{R}$, $[-10, 10]$
Coefficient matrix for VEC	Γ	$(K \times K(p-1))$	$\mathbf{R}$, $[-10, 10]$
Output			
Time series	y	$(p \times T, K)$	–

2.6.2 Definition of input and output variables

In Figure 2.12 one can see different input components (e.g. number of observations, number of lags) and an output component for scrolling the computed time series. They are connected to variables (data models) that store the values. The variables are summarized in Table 2.3. This table also shows how the validation algorithm for each input variable is configured.

These variables are centrally administered by a GaussSymbolTable. It is possible to configure the input-output components in such a way that they try to reference a local GaussSymbolTable (by default the query is made to a

non-local GaussSymbolTable). This is done here. A local GaussSymbolTable is provided by the class LocalGaussPanel. It is used here as the container that arranges all graphical program elements. Setting up local statistical data is advantageous when developing beans that want to use variable names locally or if certain variable names are already used by other parts of the application.

2.6.3 GUI components

The program *Multiple Time Series* has the following (visible) GUI components:

- The GaussFrame (the application window),
- Four GaussDataTextFields (for K, T, p, and r),
- Four JLabels (to name the text fields),
- One GaussDataTable (for y),
- One GaussDataTableScrollpane (for scrolling y),
- One EquationTermLHS (for displaying the left hand side of equations (2.1) and (2.2)).
- One EquationTermLagged (for $A_1, \ldots, A_p$ and $\Gamma_1, \ldots, \Gamma_{p-1}$),
- One EquationTermCI (for α and β),
- One EquationTermDefault (for displaying the residual term),
- One JCheckBox (for switching between equation (2.1) and (2.2))
- Three JButtons (for computing, plotting, and saving).

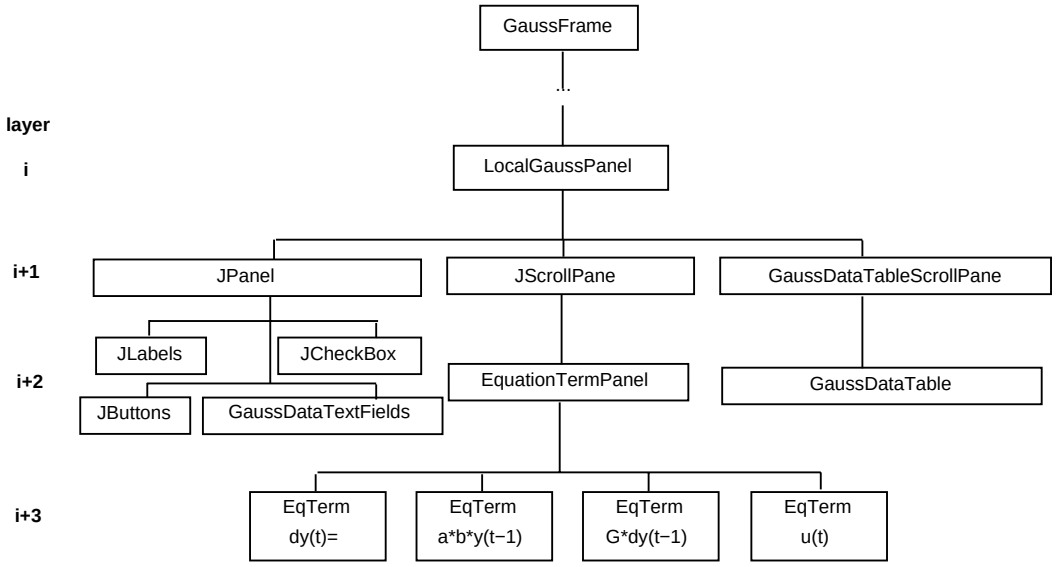


Figure 2.13: Containment hierarchy.

2.6.4 The containment hierarchy

All visual (and nonvisual) components are hold by so called *containers*. This is a component that can contain child components. The container can be associated with a so called *layout manager* that is responsible for arranging all child components with respect to layout constraints, e.g. to display the label *Length of series* in the upper left corner. Here, a LocalGaussPanel is used as the container which holds all input and output components of the previous section. The LocalGaussPanel is also a child, added to some other parent component. It represents the i-th layer in the containment hierarchy of the application. Three more layers are added to the LocalGaussPanel. In the layer $i+1$ are two non visual components, a JPanel and a JScrollPane. The JPanel is introduced for convenient layout management. The JScrollPane becomes visible if the EquationTerms in the EquationTermPanel grow to large.

Statistical data is administered by the GaussSymbolTable associated with the LocalGaussPanel.

2.6.5 Event handling

User actions (events) cause program behavior. This is achieved by linking an event listener to a program action. Every time a key is typed or a mouse button is pushed, an event occurs. Any object can be notified of the event. All the object has to do is to implement the appropriate listener interface and register as an event listener on the appropriate event source (i.e. the GUI components).

Event-handling code executes in a single thread, the event-dispatching thread. This ensures that each event handler will finish executing before the next one starts executing. For instance, the request to compute the time series in the example executes in the event-dispatching thread. Painting code also executes in the event-dispatching thread. This means that while the computation request is executing, the program's GUI is frozen – it won't repaint or respond to mouse clicks, for example. However, the program should not wait until the time series is computed and communicated from GAUSS to *Multiple Time Series*.

In the computation request method only a GaussThread with the GAUSS code is created and registered with the next `gauss.Gauss` object. The real computation is then executed in the "background" without blocking the GUI. This ensures that the program's perceived performance stays acceptable.

Action listener

Every JButton has an action listener. Whenever the button is clicked, the action listener calls the specified method. In Figure 2.14 from each button an arrow leads to a method that executes some code. These arrows symbolize that "something" is "listening" to the buttons and execute the given methods once the button is clicked (an action is performed), e.g execute the method `computeTimeSeries()` when the *Compute series*-button is clicked.

Table model listener

The GaussDataTextFields are not connected with an action listener. Due to its specialization the GaussDataTextField tries to store every new input to the data model (a GaussData object) whenever the return key is hit or the input focus is moved away. Storage is done if the outcome of the input validation was positive.

Therefore, it is better to connect GaussDataTextFields with TableModelListeners. Changes in the underlying GaussData object are notified to all interested listeners. When the process dimension K is changed in the program *Multiple Time Series*, the parameters A, α, β, and Γ must be checked. If p changes, the parameters A, and Γ must be checked. When r changes, α and β must be checked. In all cases y is no longer valid and should be reset.

Window listener

A window listener that is registered on the application frame triggers a confirm-exit dialog and, eventually, terminates the application when the window is closed. This listener is already a feature of the GaussFrame. It is not necessary to implement it for every new application that uses a GaussFrame as the application window.

2.6.6 Visual programming

Figure 2.14 is an exact snapshot from the VCE that was used to create the bean which handles user input and computation.

The visual composition editor arranges the visual components exactly as they appear in the final program. The configuration of the components can turn out to be complex. The configuration includes, for example, the position of a component relative to other components, its color, text, font, etc. Tools

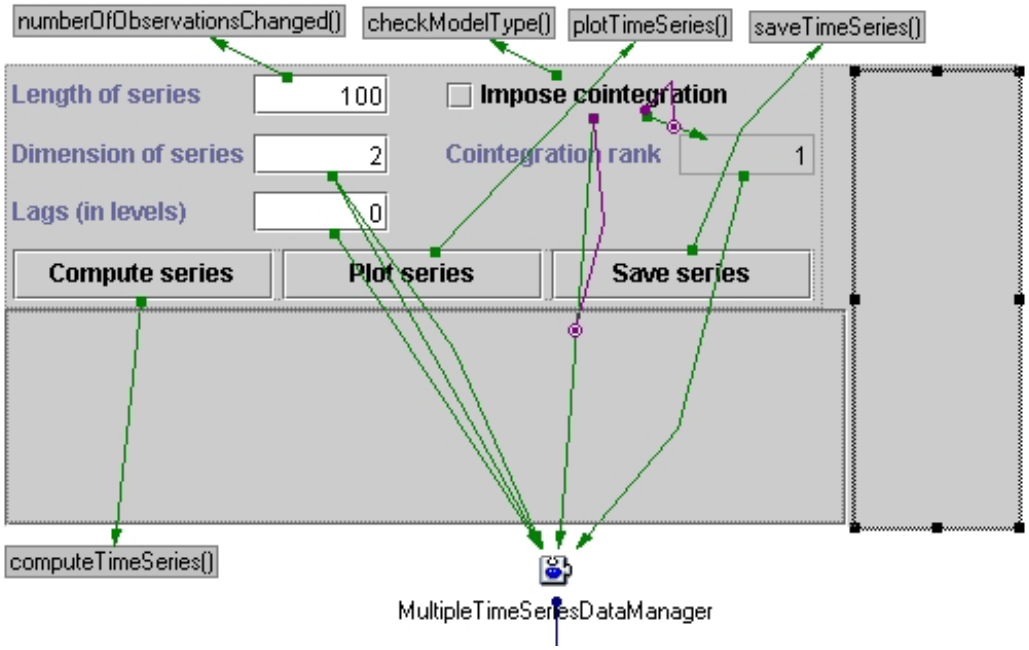


Figure 2.14: Visual representation of the GUI bean for the program *Multiple Time Series*. Note the arrows which symbolize software behavior.

such as the property sheet in Figure 2.11 help to change the default configuration of a software component fast and reliable. Time consuming search in the textual code is therefore avoided.

In addition to the (static) configuration of software components it is also possible to picture the (dynamic) program logic. The behavior of the software can be visually determined. It is possible to connect a specific user action (e.g. click on mouse button) with a method call. This is represented by the arrows. For example, in Figure 2.14 an arrow leads from the box "Compute series" to the box "computeTimeSeries()". This arrow symbolizes the internal execution of the software method computeTimeSeries() if the user clicks on the button "Compute series". It does not matter where the executed methods are located. In most cases they are defined in the GUI class. Non GUI related behavior should be placed in special classes, for example the bean Multiple-

TimeSeriesDataManager in Figure 2.14 provides computational services that are not directly related to the GUI.

2.7 Summary and perspectives

The approach outlined in this chapter can be used to build graphical user interfaces for statistical procedures which are coded in GAUSS. Numerical implementations of new econometric methods can be furnished with a GUI that supports all features of modern user interfaces. These interfaces make the procedures easier to apply in daily work. People unfamiliar with GAUSS can also work with the implemented methods by using the GUI. It can be viewed as a way to provide scientists and students easy access to recently developed statistical methods which are not found in commercial statistical software packages.

Some statistical software packages have a communication interface to other programs, e.g. Mathematica[8], or GAUSS with Mercury. The software Mercury (Breslaw (2002)) provides access to GAUSS. The difference to this work is that the current version (Mercury 4.0) only communicates with the program GAUSS Engine[9]. But the communication software developed in this work connects to GAUSS 3.2 or higher instead. The main focus of Mercury is to provide an interface between GAUSS and other Windows applications. In addition to this task the concept which had been developed here is concerned with creating menu-driven Windows applications for statistical analyses.

Only few statistical programs have tools for creating graphical control elements that query user input and control program flow. If they are available

[8]There is a toolkit that integrates Mathematica and Java. See http://www.wolfram.com/solutions/mathlink/jlink/

[9]The GAUSS Engine is a dynamic library that can be linked to other programs. See http://www.aptech.com/s2_ge.html

they only support basic GUI functionality. Furthermore, a close combination of the GUI part (such as program flow) and statistical computation cannot be avoided. Although it is possible to create complex programs when mixing GUI functionality and statistical computation (e.g. the software MulTi from Haase et al. (1992)) this approach is inefficient. However, it is the only possible approach if the operating system does not allow for multi-tasking. Nevertheless, it is complicated to extend existing menus and to add in new program parts. The user interface and the estimation routines are hard to maintain.

The approach introduced in this chapter does not suffer from these drawbacks because it exploits the advantages of multi-tasking that comes along with recent PC operating systems. The GUI program and the statistical computation are strictly separated. They could even run independently. The GUI application is not tailored to a specific statistical software package (although currently it only uses GAUSS), and the statistical procedures called are also running perfectly outside the GUI application.

Designing an effective GUI will take considerable effort. It is understood that there is a strong interest in keeping this effort low. Therefore, programming tools and techniques were discussed which facilitate development of GUIs. It is argued that widely available standard tools, such as an integrated development environment with visual programming, should be used. The level of support introduced by these tools is considerable. Visual programming makes the code of the GUI program more transparent and extremely easy to maintain. The support can be further enhanced by well designed configuration tools. The development of such configuration tools is beyond the scope of this work. But theoretically, they could enable the creation of GUIs for statistical procedures purely visually, i.e. without any textual coding.

To summarize the discussion it can be said that the strength of the concept is its component related design which allows a high degree of reusability,

extendibility and maintainability. It can be used to create single GUIs for a single procedure or to create complex menu-driven programs for a set of related statistical procedures. These procedures are connected by the GUI program. Thus different steps in the statistical analysis of a problem can be modeled with menu-driven applications. An example for a large menu-driven program is given in the next chapter. It explains and demonstrates parts of the software JMulTi.

Chapter 3

Analyzing VAR models with JMulTi

Many econometric analyses are carried out with vector autoregressive (VAR) models. The advantage of these models is that they allow the embedding of interesting economic hypotheses in a general statistical framework (Johansen (1995)) and that their statistical analysis can be done with standard methods (Lütkepohl (2000)). There already exists a comprehensive body of methods for detecting the statistical properties of time series, for model specification, model estimation and validation, for forecasting, and for model interpretation.

Due to the popularity of these models much research has been conducted in order to solve known problems of current methods. For example, there are many important macro economic time series with a structural break. These breaks may be known, for example, the European or German monetary union, or the transformation process of Eastern European economies. It is known that unit root tests that do not account for these breaks may suffer from a loss of power. Research on unit root tests for time series with a break point was carried out, for example in Lütkepohl et al. (2001).

However, recent methods are usually not provided by commercial econo-

metric software packages. In order to use a new promising method in empirical research one would have to ask the authors who published a new method for the software procedure. Even if it is available, it still can be burdensome to apply the procedure to a problem at hand. But finally, it only stays an isolated application.

This Chapter introduces the software JMulTi. It is a menu-driven program for analyzing univariate and multivariate time series. JMulTi is based on the concept explained in Chapter 2. The program consists of a graphical user interface (GUI) that executes procedures in GAUSS for Windows. The procedures implement well established and new econometric tools for time series analysis.

Extending the program structure of JMulTi is a built in feature. Program extensions are possible conditional on the existing set of econometric methods, or by providing a complete new analysis tool. Therefore, JMulTi can be seen as a framework in which new methods for the analysis of time series can be incorporated. Every new computation can be based on the results of previous procedures calls. Therefore it is possible to link many statistical procedures easily.

JMulTi can be used subject to the license agreement of Appendix A. Note the disclaimer at the end of Appendix A. JMulTi is free of charge and comes with no support.

This Chapter explains the analysis capabilities of JMulTi in the context of vector autoregression, and documents the parts provided by the author. The whole Chapter refers to JMulTi version 1.96 beta in conjunction with GAUSS for Windows version 3.2.29. This JMulTi-version can be downloaded from http://www.jmulti.de/download.html (without GAUSS). Since JMulTi has been continuously developed and improved it is worth to use the most recent version which is available from http://ise.wiwi.hu-berlin.de/oekonometrie/

`index.html`

This Chapter is organized as follows. The first section makes general remarks on program structure and handling. The steps of the VAR analysis implemented by the author are documented in the second section. The third section demonstrates JMulTi's capabilities of computing bootstrap confidence intervals for impulse responses. This will be illustrated by different models of a German monetary system.

3.1 Introduction to JMulTi

3.1.1 Program structure

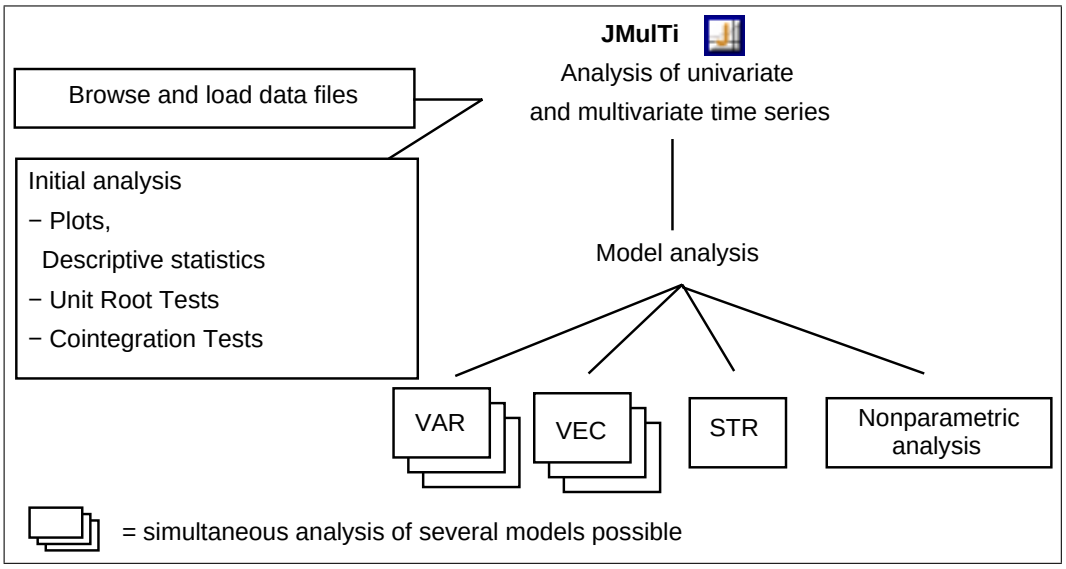


Figure 3.1: Program structure of JMulTi

The general structure of JMulTi is pictured in Figure 3.1. In addition to the browse and load tools, different tools for analyzing time series are offered. Currently JMulTi consists of analysis tools for

- Initial analysis (plots and transformation of time series, unit root tests, cointegration tests),

- VAR analysis (modeling stationary, stable processes),
- VEC analysis (modeling cointegrated processes),
- Smooth transition regression (STR) analysis (Teräsvirta (1998)),
- Non-parametric analysis (Tschernig and Yang (2000)).

3.1.2 Installation

In order to run JMulTi the software GAUSS for Windows version 3.2, 3.5, or 3.6 and a Java interpreter must be installed on the computer. GAUSS is a commercial statistical software package. See http://www.aptech.com for details. Regarding Java, it is recommended to use version 1.3.1. The larger installation routine will install this Java version if it is desired. JMulTi runs on Windows 98, 2000, ME, NT4.0.

The installation process is standard. At first, a copy of the installation routine must be obtained from http://ise.wiwi.hu-berlin.de/oekonometrie/engl/indexeng.html following the *Software*-link. The larger of the two setup files contains the recommended Java run time environment (JRE), the smaller one comes without the JRE.

The installation process is started by executing the downloaded file. All necessary information and instructions will be displayed. The installation is fast and will take about one or two minutes.

3.1.3 Starting JMulTi

The program is started by a double click on the JMulTi icon. A window like the snap shoot shown in Figure 3.2 (henceforth called *main window*) appears. The area within the main window is organized as a desktop. It may contain further (internal) windows (henceforth *sub windows*) each representing a JMulTi tool

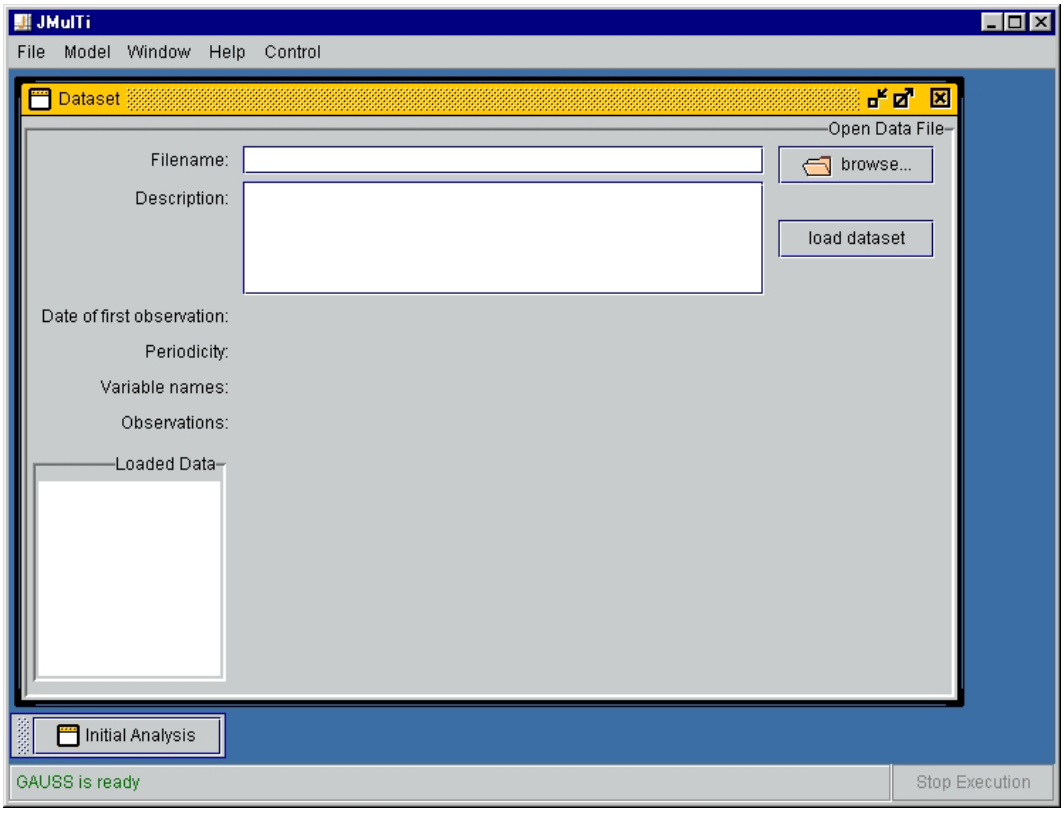


Figure 3.2: Main window of JMulTi with opened Dataset sub window and iconified Initial Analysis sub window.

of Figure 3.1. Sub windows can be iconified, restored/maximized, or closed by the

buttons in the upper right corner of the window. If the closing operation will delete analysis results the user is asked for confirmation. Furthermore, sub windows can be resized. However, all sub windows have a minimum size which cannot be changed.

After program start two sub windows are opened, one for loading data from disk into the program and one for conducting initial analyses. Further sub windows can be added during the analysis.

3.1.4 General program handling

The main window has a menu bar with the following menus:

File Program exit and loading data sets.

Model Opens sub windows for VAR, VEC, STR, and non-parametric analysis.

Window Contains a list of all currently available sub windows. This list always contains the sub windows Dataset and Initial Analysis. Model windows are added (removed) as soon as they are created (closed). This menu allows to switch conveniently between different sub windows.

Help Access help functions.

Control Offers tools for controlling the computing environment. There are functions for restarting GAUSS, for displaying a window containing all system messages, and for displaying a window that shows all variables that are available JMulTi-wide.

Sub windows may also contain menus for accessing different analysis methods. Their contents are discussed later. A sub window menu is disabled (i.e. it cannot be used) if all of its entries are disabled. During model analysis, many menu entries become enabled. For instance, in the VAR analysis, the impulse responses can only be analyzed (i.e. the respective menu item becomes enabled) if the model has been estimated.

In addition to the menus in the main and sub windows there are different elements in the GUI that open pop-up menus. For example, the time series list or various coefficient matrices can open a pop-up menu. It is triggered by clicking the right mouse button. It displays a number of context related actions. The available pop-up menus are discussed later, too.

In the following menu selections are referred to as *select* *menu→option* *from some window*, e.g. in order to open the load data set tool, select *File→Dataset* from the main window.

At different points the user is asked to make a selection from a list of items, e.g. to select some variables. A single item is selected by a simple mouse click on that item. If several items must be selected one can make a so called single interval selection or a multiple interval selection. A single interval selection is made by clicking on the first item, then holding down the Shift-button and clicking on the last item. A multiple interval selection is made by holding down the Ctrl-button and making various single selections. It is also possible to unselect items by doing multiple interval selection on selected items.

3.1.5 Loading data sets

In Figure 3.2, the sub window Dataset can be seen which offers basically two actions: Browsing data files and loading data sets. The browse function allows to view data files before adding the data to the list of time series. When the correct file is found its contents are loaded with the load dataset button. The new time series are added to the previously loaded time series. Throughout JMulTi, a time series is identified by its name. If a new time series has a name that is already in use, JMulTi asks whether or not the existing data should be overwritten. After loading, the sub window Dataset can be minimized or closed. It can be opened again by selecting *File→Open Dataset* or *Window→Dataset* from the main window.

A valid data file contains one or more time series and must consist of ASCII symbols only. Text between /* and */ is interpreted as a comment. Comments can be placed anywhere in the data file. The first non-comment row contains some general information about the data set. It has three entries: The first is the number of time series (number of data columns). It is followed by the

date of the first observation. The third entry indicates the periodicity of the observations, i.e. the number of subperiods within one principal period. For instance, for quarterly data a year is the principal period, which has 4 subperiods. Therefore, 1960.1 denotes the first quarter of year 1960 for quarterly data (periodicity=4), October of 1960 for monthly data (periodicity=12), and week 10 of 1960 for weekly data (periodicity=52). The periodicity must be an integer value ≥ 1, but otherwise it is not restricted to certain values as long as it is consistent with the subperiod in the date of the first observation.

The next (non-comment) line contains, separated by blanks, the names of the time series. Each name should be used only once. The remaining (non-comment) lines contain the observations for the time series, beginning with $t = 1$. The observations in a line are, again, separated by blanks. A file with 8 time series could look as follows:

```
/*
some comments
....
*/
8  1960.1  4
lm1  lp  lyr  rl  lpim  s1  s2  s3
3.8497  3.4038  5.4134  0.0619  3.8269  1    0    0
3.8786  3.3802  5.4946  0.0640  3.8152  0    1    0
3.8917  3.4362  5.5761  0.0640  3.7957  0    0    1
3.9596  3.4198  5.5915  0.0619  3.7927  0    0    0
...
```

The data file can contain one or more missing values. They are identified by the symbol `NaN` at the respective position. For instance, the above example with missing values for the whole 4th quarter 1959 would be:

```
/*
some comments
....
*/
8  1959.4  4
lm1  lp  lyr  rl  lpim  s1  s2  s3
```

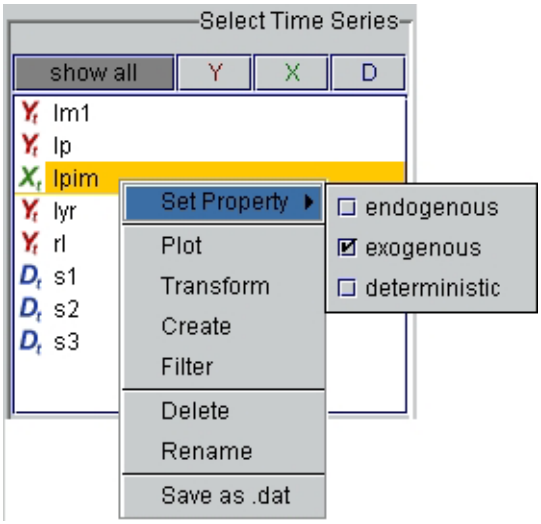


Figure 3.3: Time series selection list

```
NaN     NaN     NaN     NaN     NaN     NaN  NaN  NaN
3.8497  3.4038  5.4134  0.0619  3.8269  1    0    0
3.8786  3.3802  5.4946  0.0640  3.8152  0    1    0
3.8917  3.4362  5.5761  0.0640  3.7957  0    0    1
3.9596  3.4198  5.5915  0.0619  3.7927  0    0    0
...
```

Note, that the date of first observation is now 4th quarter 1959, and not 1st quarter 1960 as in the first example.

In addition to the name, a time series is assigned an endogenous, exogenous, or deterministic *property*. It is expressed with the symbols Y, X, and D in front of the name of the time series in the time series list (see Figure 3.3). This "property" is important when selecting variables for a specific model but does not describe any statistical properties of the time series. All time series containing only zeros and/ or ones are regarded *deterministic*. All other time series are regarded *endogenous*. In the sample data file above, the first five columns are interpreted as endogenous variables whereas the last three columns are interpreted as deterministic variables. The property of a variable can be changed in any time series selection list with the *Property* option of its

associated popup menu. In Figure 3.3, the property of the time series *lpim* was changed to exogenous (X).

In addition to the *Set Property* option, there are a number of other important entries in the pop up menu of the time series selection list, for example the *Transform* and *Rename* options. The *Transform* option allows to apply some standard transformation to the selected time series. The transformed series is stored in a new time series which is added to the time series list. The name of a time series can be changed with the *Rename* option.

3.2 Analysis of VAR models

This section describes the VAR modeling framework of JMulTi. It starts with an enumeration of JMulTi's tools for determining the statistical properties of time series. Then, the functional forms of the VAR and VEC models used are discussed. How these models are specified is explained next. Part of the specification are subset constraints on model coefficients. The estimation procedures for the VAR and VEC models are treated in separate sub sections. The sub section on structural analysis explains in detail JMulTi's tools for the interpretation of estimated VAR models and the thereby used inferential procedures.

Any statistical model should be carefully checked for the assumptions it imposes. JMulTi offers various residual based tools for checking the assumptions of the fitted VAR or VEC models, for example diagnostic tests, residual plots and correlation analysis. Since they were not contributed by the author they are not documented here. The same holds for the forecast tool of JMulTi.

3.2.1 Determining statistical properties of time series

The sub window Initial Analysis contains a collection of methods for exploring the statistical properties of the loaded time series. There are three general categories

Workbench	Various plot functions, computing descriptive statistics, and creating new time series,
UR Tests	Different unit root tests, and
Cointegration Tests	Different cointegration tests.

Unit root tests are tools for determining the order of integration of a time series. Stationary time series have time invariant means, variances, and covariance structure. They are integrated of order 0, or short $I(0)$. In JMulTi, stationary series can be analyzed with VAR models. However, many economic time series do not have this property. If they have trends which can be removed by differencing, i.e. y_{it} is not stationary while $\Delta y_{it} = y_{it} - y_{i,t-1}$ is, they are called integrated of order 1. In general, nonstationary series which are stationary after differencing them d times are called $I(d)$.

If multiple series are $I(1)$ then the next step is to test whether these series are cointegrated. In general, $I(d)$ variables is called cointegrated if there exists a linear combination which is $I(d_1)$ with $d_1 < d$. In JMulTi, cointegrated series can be modeled with the VEC model. It is also possible to fit a VAR model ignoring the property of cointegration.

3.2.2 The VAR model

The general specification of VAR models in JMulTi is

$$A_0 y_t = AY_{t-1} + BX_t + Cd_t + v_t \tag{3.1}$$

where $y_t = (y_{1t}, \ldots, y_{Kt})'$ is a K–dimensional random vector, containing the endogenous variables, A_0 is a non-singular $(K \times K)$ matrix with unit diagonal, modeling the structural form,

$$AY_{t-1} = A_1 y_{t-1} + \ldots + A_p y_{t-p}$$

is the autoregressive term where the A_i are $(K \times K)$ coefficient matrices,

$$BX_t = B_0 x_t + B_1 x_{t-1} + \ldots + B_q x_{t-q}$$

is the exogenous term, where $x_t = (x_{1t}, \ldots, x_{Lt})'$ is a L dimensional (stochastic) exogenous variable, and B_i are $(K \times L)$ coefficient matrices,

$$Cd_t$$

is the deterministic term, where d_t is a M dimensional non stochastic variable, and C is its $(K \times M)$ coefficient matrix, and $v_t = (v_{1t}, \ldots, v_{Kt})'$ is a white noise process with $E(v_t) = 0$, $E(v_t v_t') = \Sigma_v$, and $E(v_t v_s') = 0$ for $s \neq t$ (see Lütkepohl (1993a)). If $A_0 \neq I_K$ model (3.1) is called structural form.

The reduced form is obtained by left multiplying (3.1) with A_0^{-1}

$$\begin{aligned} y_t &= A_0^{-1} AY_{t-1} + A_0^{-1} BX_t + A_0^{-1} Cd_t + A_0^{-1} v_t \\ &= \overline{A} Y_{t-1} + \overline{B} X_t + \overline{C} d_t + u_t \qquad (3.2) \\ &= \overline{G} Z_t + u_t \qquad (3.3) \end{aligned}$$

$$\text{with} \quad \overline{G} = [\overline{A}, \overline{B}, \overline{C}] \quad \text{with dimension} \quad (K \times N),$$

$$N = Kp + L(q+1) + M$$

$$Z_t = \begin{bmatrix} Y_{t-1} \\ X_t \\ d_t \end{bmatrix}.$$

This can be written more compactly as

$$Y = \overline{G} Z + U \qquad (3.4)$$

where $Y = [y_1, \ldots, y_T]$, $Z = [Z_1, \ldots, Z_T]$, and $U = [u_1, \ldots, u_T]$.

Currently, JMulTi offers only the analysis of VAR models in reduced form (3.2). The VAR modelling is started by selecting *Model→VAR* from the main frame. A new sub window appears in the desktop area of the main window.

3.2.3 The VEC model

If cointegration relations are present it is useful to consider a parameterization which incorporates the cointegration structure. Vector error correction (VEC) models allow for such parametric structure. See, for example, Lütkepohl (1993a, Chapter 11) for different ways of including cointegration relations in the VAR model.

The functional form of the VEC model in JMulTi is

$$F_0 \Delta y_t = \alpha ec_{t-1} + F \Delta Y_{t-1} + B X_t + C^{sys} d_t^{(sys)} + v_t \tag{3.5}$$

where $\Delta y_t = y_t - y_{t-1}$ is the first difference of y, F_0 a $(K \times K)$ matrix of structural coefficients, α is a $(K \times r)$ matrix of coefficients, ec_{t-1} is the $(r \times 1)$ dimensional error correction term, that models the r cointegration relations by

$$ec_{t-1} = \beta y_{t-1} + \beta^{(d)} d_t^{(ec)}, \tag{3.6}$$

where β and β^d are coefficient matrices with dimension $(r \times K)$ and $(r \times M^{(d)})$, respectively, and $[\beta, \beta^d]$ contains the r cointegration vectors,

$$F \Delta Y_{t-1} = F_1 \Delta y_{t-1} + \ldots + F_{p-1} \Delta y_{t-p+1}$$

is the term of differenced lagged endogenous variables, where F_i are $(K \times K)$ coefficient matrices, $C^{(sys)}$ is a $(K \times M^{(sys)})$ coefficient matrix of the $M^{(sys)}$ dimensional vector of deterministic variables $d^{(sys)}$. The terms BX_t and v_t are defined as for the VAR model (3.1).

In order to model ec_t optionally with deterministic terms, the elements in the variable d_t of equation (3.1) are split into $d_t^{(sys)}$ and $d_t^{(ec)}$ in (3.5) and (3.6). The notion that ec at $t-1$ is modeled by $d^{(ec)}$ at t can be a source of misunderstanding at first sight. In some cases this modeling is not important, e.g. if $d_t^{(ec)}$ just contains a constant term. If ec_{t-1} is modeled with an additional step dummy, i.e.

$$d_t = \begin{cases} 0 & \text{if} \quad t < t_1 \\ 1 & \text{if} \quad t \geq t_1 \end{cases}$$

the time index of course matters and d_t is usually split up as $d_t = \Delta d_t + d_{t-1}$. The ec_{t-1} is modeled with $d_t^{(ec)} = d_{t-1}$ and $d_t^{(sys)} = \Delta d_t$ is added to the short term dynamics. Due to ease of notion the exposition of (3.6) is chosen here but a cautious design and selection of deterministic variables is necessary.

The reduced form VEC model is obtained by left multiplying (3.5) with F_0^{-1}

$$\begin{aligned} \Delta y_t &= F_0^{-1}\alpha ec_{t-1} + F_0^{-1}F\Delta Y_{t-1} + F_0^{-1}BX_t + F_0^{-1}C^{(sys)}d_t^{(sys)} + F_0^{-1}v_t \\ &= \overline{\alpha} ec_{t-1} + \overline{F}\Delta Y_{t-1} + \overline{B}X_t + \overline{C}d_t + u_t \qquad (3.7) \\ &= \overline{G}Z_t + u_t \qquad (3.8) \end{aligned}$$

$$\text{with} \quad \overline{G} = [\overline{\alpha}, \overline{F}, \overline{B}, \overline{C}^{(sys)}]$$

$$Z_t = \begin{bmatrix} ec'_{t-1} \\ \Delta Y_{t-1} \\ X_t \\ d_t^{(sys)'} \end{bmatrix}.$$

Again, this can be written more compactly as

$$Y = \overline{G}Z + U \qquad (3.9)$$

where $Y = [\Delta y_1, \ldots, \Delta y_T]$, $Z = [Z_1, \ldots, Z_T]$, and $U = [u_1, \ldots, u_T]$.

Models (3.1) and (3.5) are indeed equivalent:

$$
\begin{aligned}
F_0\Delta y_t &= \alpha ec_{t-1} + F\Delta Y_{t-1} + BX_t + C^{sys}d_t^{(sys)} + v_t \\
\Longleftrightarrow & \\
F_0(y_t - y_{t-1}) &= \alpha(\beta y_{t-1} + \beta^{(d)} d_t^{(ec)}) \\
&\quad + F_1(y_{t-1} - y_{t-2}) + \ldots + F_{p-1}(y_{t-p+1} - y_{t-p}) \\
&\quad + BX_t + C^{sys}d_t^{(sys)} + v_t \\
\Longleftrightarrow & \\
F_0 y_t - F_0 y_{t-1} &= \Pi y_{t-1} + \Pi^{(d)} d_t^{(ec)} \\
&\quad + F_1 y_{t-1} - F_1 y_{t-2} + \ldots + F_{p-1} y_{t-p+1} - F_{p-1} y_{t-p} \\
&\quad + BX_t + C^{sys}d_t^{(sys)} + v_t, \\
&\qquad \text{with} \qquad \Pi = \alpha\beta \\
&\qquad\qquad\quad \Pi^{(d)} = \alpha\beta^{(d)} \\
\Longleftrightarrow & \\
F_0 y_t &= (F_1 + F_0 + \Pi) y_{t-1} + (F_2 - F_1) y_{t-2} + \ldots \\
&\quad + (F_{p-1} - F_{p-2}) y_{t-p+1} - F_{p-1} y_{t-p} \\
&\quad + BX_t + [C^{sys}, \Pi^{(d)}] \begin{bmatrix} d_t^{(sys)} \\ d_t^{(ec)} \end{bmatrix} + v_t \\
\Longleftrightarrow & \\
A_0 y_t &= A_1 y_{t-1} + A_2 y_{t-2} + \ldots + A_p y_{t-p} \\
&\quad + BX_t + C d_t + v_t, \\
&\qquad \text{with} \quad A_0 = F_0 \\
&\qquad\qquad\quad A_1 = F_1 + F_0 + \Pi \\
&\qquad\qquad\quad A_i = F_i - F_{i-1}, \quad i = 2, \ldots, p-1 \\
&\qquad\qquad\quad A_p = -F_{p-1} \\
&\qquad\qquad\quad C = [C^{sys}, \Pi^{(d)}] \\
&\qquad\qquad\quad d_t = [d_t^{(sys)'}, d_t^{(ec)'}]'
\end{aligned}
$$

A cointegration analysis is started by selecting *Model→VEC* from the main

window. A new sub window will appear within the main window for specifying and analyzing model (3.5) or (3.7).

3.2.4 Model specification

Model specification deals with variable selection (y, x, d) and lag selection (p, q). In the cointegration analysis the cointegration rank r is specified additionally and the error correction term ec_t can be modelled optionally. It is also possible to analyze structural VEC models. Declaring zero restrictions for single elements of model coefficient matrices can be viewed as a part of the model specification process. However, this is treated in the next sub sections.

Selection of variables

When the VAR and VEC sub windows are displayed the first time, the variables and lags input area is shown. From any later step in the analysis one can return to this display by selecting *Specification→Specify Model* from the VAR or *Specification→Select Variables, Lags, Rank* from the VEC sub window.

VAR models must contain at least one, VEC models at least two endogenous variables. The selection is made with the variable selection list, see Figure 3.3. In addition one can select exogenous and deterministic variables. Standard deterministic variables, such as a constant term, seasonal dummies and a trend, can be added with the check boxes. Initially, all models contain a constant term. Unselect the check box if this is not desired.

The selection order determines how the time series are stacked in the variables y, x, and d. The start and end date are set such that the largest possible sample size is used. If the dates are changed to other valid values, the sample size adjusts.

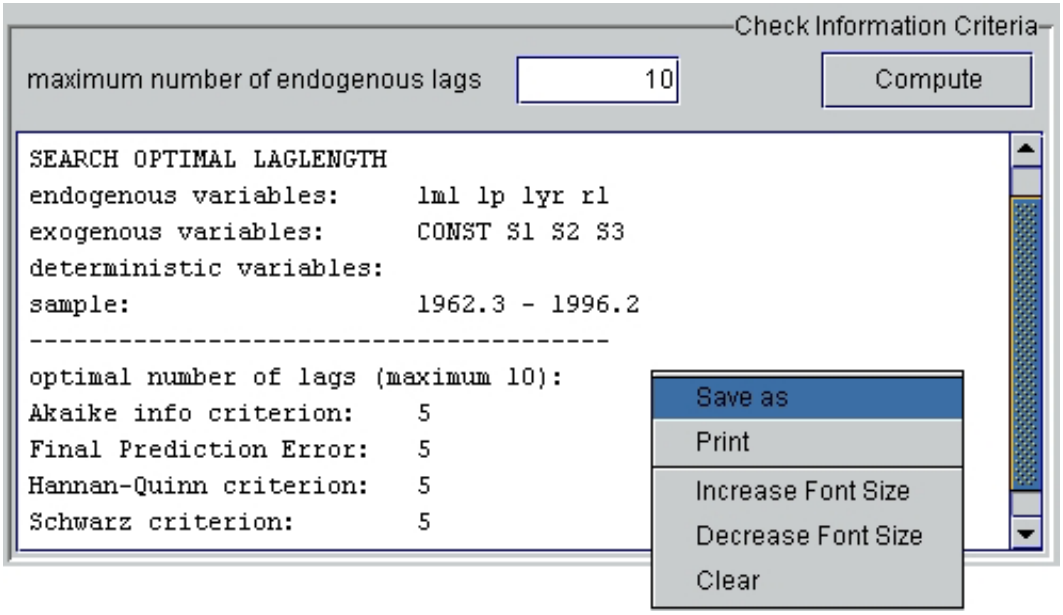


Figure 3.4: Computing information criteria for a VAR model, note again the opened popup menu

Number of lags and cointegration rank

After selecting endogenous (exogenous) variables the number of lags p (q) can be edited. For specifying the number of lags for the endogenous variables the FPE, AIC, HQ and SC criteria (see Lütkepohl (1993a, Ch. 4)) can be computed. Note that in the VEC analysis the optimal number of lags refers to the lagged differences, not levels. In the VEC analysis, the first input mask contains additionally a field for editing the cointegration rank r, $0 < r < K$, which is set to $r = 1$ by default.

Modeling the cointegration relation(s)

Cointegration relations can only be modeled in the VEC analysis. There are three points in the VEC sub window that affect the modeling of the error correction term. The cointegration rank r is edited by selecting *Specification→Select variables, lags, rank* from the VEC sub window. This models $ec_{t-1} = \beta y_{t-1}$.

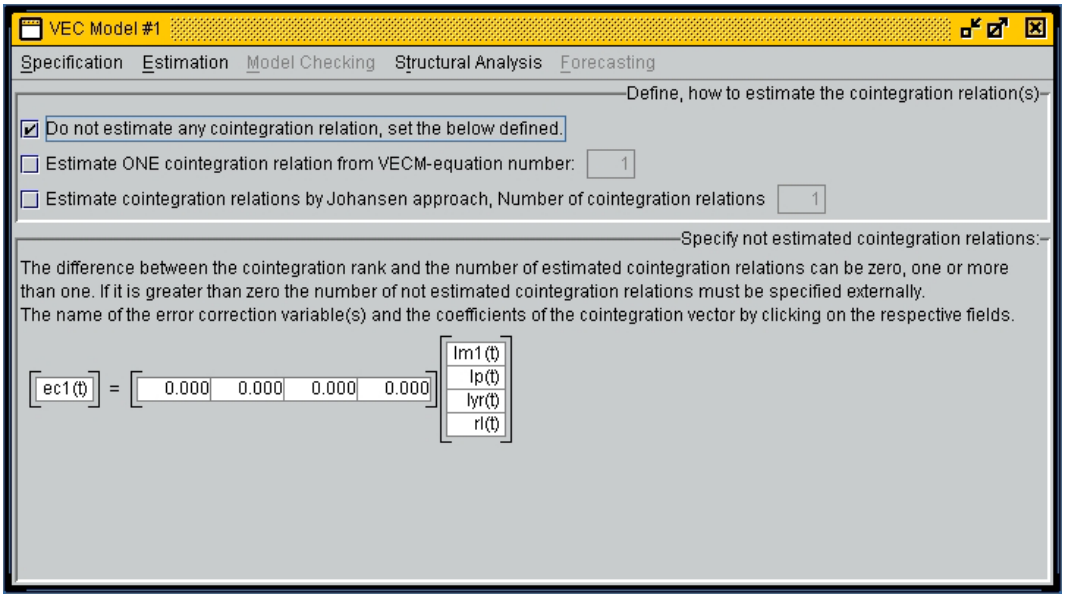


Figure 3.5: Specification of cointegration relation(s)

Calling *Specification→Add Deterministics to Co-int Relation* allows to restrict deterministic variables to the cointegration relation. From the set of included deterministic variables one can select none, some, or all to enter the modeling of ec_{t-1}, see equation (3.6).

Finally, in *Specification→Specify Estimation of Co-int Relation* it is possible to determine the number of cointegration relations r_1 to be estimated, with $(0 \leq r_1 \leq r)$, and the method for estimating the r_1 cointegration vectors which is applied in the first step of the two step estimation strategy (see below). Figure 3.5 shows a screenshot of this input mask. Possible options are

1. No estimation $(r_1 = 0)$,
2. Estimation of $r_1 = 1$ cointegration vector from an equation of the VEC by OLS,
3. Estimation of $0 < r_1 \leq r$ cointegration vectors by the Johansen approach.

In case of $r_1 < r$, the $r - r_1$ cointegration vectors which are not estimated

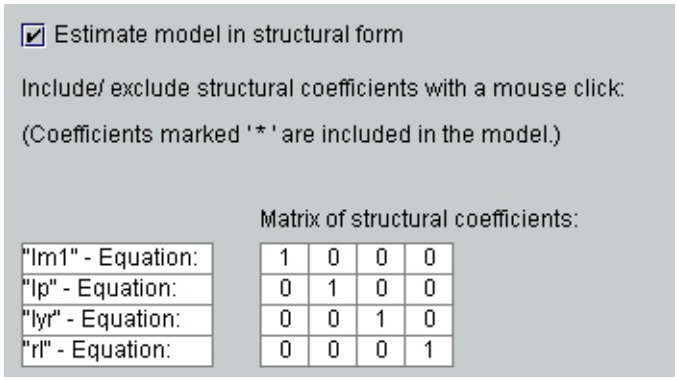


Figure 3.6: Specification of structural form

must be specified manually. In this case a convenient input structure appears as shown in the lower part of Figure 3.5.

Structural form modeling

Currently, structural forms can only be modeled in the VEC analysis. Structural VECs are specified by selecting *Specification→Specify Structural Form* from the VEC sub window.

At the first time $F_0 = I_K$ which can be changed with mouse clicks on the desired structural coefficients. The matrix element changes its symbol from *0* to ***. This means that the respective structural coefficient is estimated. A structural coefficient is removed with a mouse click again.

There is also a switch for estimating a structural model, represented by the check box *Estimate model in structural form*, see Figure 3.6. It can be unselected without affecting the structural coefficients specified before.

3.2.5 Parameter constraints

Linear constraints for a general coefficient matrix G of dimension $(M \times N)$ can be specified by

$$g = \text{vec}(G) = R\zeta + r \tag{3.10}$$

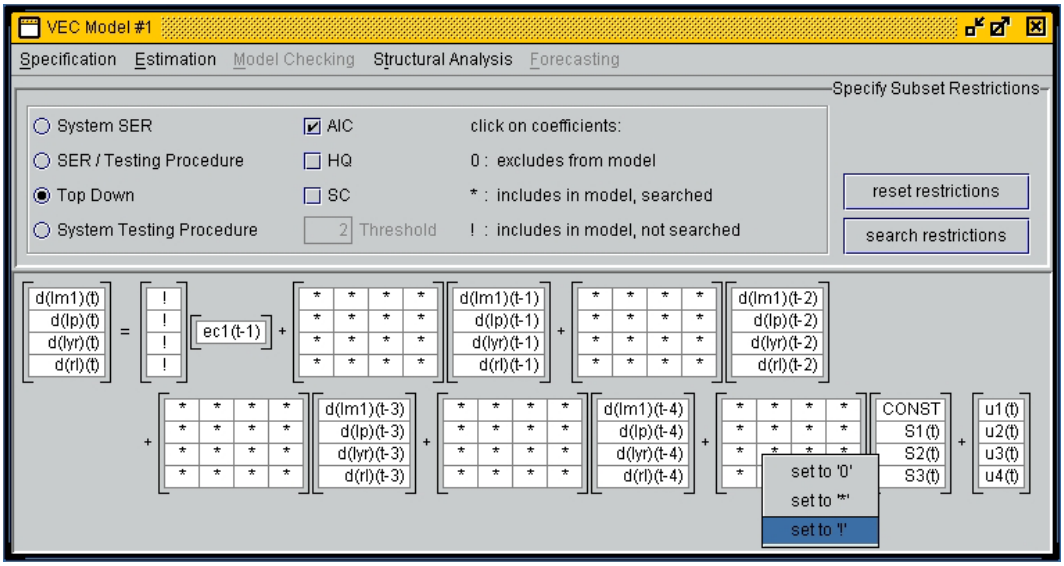


Figure 3.7: Specification of subset restrictions

where $g = \text{vec}(G)$ is a $(MN \times 1)$ vector, R is a known $(MN \times w)$ matrix of rank w, ζ is an unrestricted $(w \times 1)$ vector of unknown parameters, and r is a $(MN \times 1)$ vector of known constants (see Lütkepohl (1993a)). For instance, subset restrictions can be expressed by $r = 0$ and a matrix R that is derived from a MN-dimensional identity matrix with the i-th column removed if g_i is constrained to 0.

Another possibility to express linear constraints in G is

$$Sg = s \tag{3.11}$$

where S is a known $(v \times MN)$ matrix of rank v and s is a known $(v \times 1)$ vector. One can show the equivalence of (3.10) and (3.11) (e.g. Lütkepohl (1993a, Chapter 5)).

Both expressions will be used later when describing estimators and tests, depending on which is particularly suitable. Although the estimation algorithm can process general linear restrictions of the form (3.10) (see Appendix D), only subset constraints can be imposed with JMulTi currently.

3.2.6 Model reduction procedures

Imprecise parameter estimates in the VAR analysis is a common practical problem because the number of parameters is often quite substantial relative to the available sample size Lütkepohl (1993a, Chapter 5). Runkle (1987) comes to the conclusion that unrestricted VAR's often do not help much in order to answer interesting macroeconomic questions. The large confidence intervals for impulse responses do not allow a useful inference on these quantities. In a comment to Runkle (1987) Watson (1987) argues to use prior information in order to increase the precision of the estimates. The model reduction procedure is a type of modelling strategy which tries to better extract the information from the available data.

Model reduction procedures aim at eliminating variables for which insignificant parameters were estimated. This means to impose zero restrictions on elements of the coefficient matrices in (3.1), (3.2), (3.5), or (3.7). In JMulTi, subset models can be specified in a straightforward manner. It is possible to set the zero constraints by hand, to apply a search algorithm, or to do a mixture of both, e.g. to apply a search algorithm conditionally on already set constraints. This allows to conduct a highly individual search procedure.

In Figure 3.7, the elements of the coefficient matrices are replaced by symbols, indicating how the estimation and search algorithms treat the respective variable. The different meanings are summarized in the following table:

	Treatment of respective variable in	
Symbol	estimation	search
*	include	include
!	include	exclude
0	exclude	exclude

A mouse click on a symbol will change it in the order { ... , * , ! , 0 , * , ! , 0 , * , ... } The popup menu associated with the matrices allows to

uniformly include or exclude the whole lagged endogenous, lagged exogenous, or deterministic coefficient matrix from the estimation or search algorithm. The search algorithms applied are described and analyzed in Lütkepohl (1993a, Chapter 5, Sec. 5.2.8b) and Brüggemann and Lütkepohl (2001).

3.2.7 VAR Model estimation

Structural form VAR

Estimation of structural VARs (3.1) is currently not possible in JMulTi. Structural analysis is only available within the cointegration analysis.

Reduced form VAR

Model (3.4) is estimated by ordinary least squares (OLS) or estimated generalized least squares (EGLS). For a discussion of these estimation methods see, for instance, Judge et al. (1988) and for the application in the context of VAR models see Lütkepohl (1993a).

The following OLS estimators are computed for model (3.4) in JMulTi

$$\begin{aligned}
\widehat{\widehat{G}} &= YZ'(ZZ')^{-1} \\
\widehat{U} &= Y - \widehat{\widehat{G}}Z \\
\widehat{\not{\Sigma}}_u &= \widehat{U}\widehat{U}'/(T-N) \\
\widehat{\not{\Sigma}}_{\overline{g}} &= (ZZ'/T) \otimes \widehat{\not{\Sigma}}_u
\end{aligned}$$

where N is the number of estimated parameters in each single equation, see Equation (3.3) on page 66. For computing the standard deviation of $\widehat{\widehat{G}}$ and the respective t-ratios, the asymptotic distribution of $\widehat{\overline{g}} = \mathrm{vec}(\widehat{\widehat{G}})$

$$N\left(g, \not{\Sigma}_g/T\right)$$

is used, see Lütkepohl (1993a), where $N(\mu, \sigma^2)$ is the normal distribution with mean μ and variance σ^2. The above estimator of the residual covariance matrix,

$\widehat{\not\Sigma}_u$, will be referred to as $\widehat{\not\Sigma}_{u,OLS}$ in the following. The residuals of the reduced form are computed as

$$\widehat{u}_t = y_t - \widehat{\overline{G}} Z_t \tag{3.12}$$

where $\widehat{\overline{G}}$ can also denote other estimators of the coefficient matrix which are introduced in the following.

On the vectorized version of (3.4) linear constraints in the form of (3.10) can be imposed

$$\begin{aligned} \boldsymbol{y} &= \text{vec}(Y) = (Z' \otimes I_K)\text{vec}(\overline{G}) + \text{vec}(U) \\ &= (Z' \otimes I_K)(R\overline{g} + r) + \boldsymbol{u} \end{aligned}$$

$$\Longleftrightarrow$$

$$\begin{aligned} \boldsymbol{z} &= (Z' \otimes I_K)R\overline{g} + \boldsymbol{u} \qquad (3.13) \\ &\text{with} \quad \boldsymbol{z} = \boldsymbol{y} - (Z' \otimes I_K)r \\ &\qquad\quad \boldsymbol{u} = \text{vec}(U). \end{aligned}$$

Using (3.13), the following EGLS estimators for model (3.4) subject to linear constraints as specified in (3.10) are computed in JMulTi:

$$\begin{aligned} \widehat{\zeta} &= \left[R'(ZZ' \otimes \widehat{\not\Sigma}_{u,OLS}^{-1})R\right]^{-1} R'(Z \otimes \widehat{\not\Sigma}_{u,OLS}^{-1})\boldsymbol{z} \\ \widehat{\overline{g}} &= R\widehat{\zeta} + r \\ \widehat{U} &= Y - \widehat{\overline{G}} Z \\ &\qquad \text{with} \quad \widehat{\overline{G}} \quad \text{being the unvectorized} \quad \widehat{\overline{g}} \\ \widehat{\not\Sigma}_u &= \widehat{U}\widehat{U}'/T \\ \widehat{\not\Sigma}_{\overline{g}} &= R\left[R'((ZZ'/T) \otimes \widehat{\not\Sigma}_u^{-1})R\right]^{-1} R' \\ \text{vec}(\widehat{\overline{G}}_{EGLS}) &\xrightarrow{d} N\left(\text{vec}(\overline{G}), \not\Sigma_{\overline{g}}\right) \end{aligned}$$

For computing the standard deviation of $\widehat{\overline{G}}$ the asymptotic distribution of $\text{vec}(\widehat{\overline{G}})$

$$N\left(\text{vec}(G), \not\Sigma_{\overline{g}}/T\right)$$

is used, see Lütkepohl (1993a).

Selection of estimation method

The estimation method is selected automatically. EGLS estimators are computed if (subset) constraints were specified, otherwise the OLS estimators are computed.

3.2.8 VEC Model estimation

Structural form VEC

Currently, structural VECs can only be estimated with the two step estimation strategy (see below).

Parameters of a structural model are efficiently estimated with the three-stage least squares estimator (3SLS). The estimator and its properties are treated in detail in Dhrymes (1974, Chapter 4) and Judge et al. (1988, Chapter 14).

The 3SLS estimator is chosen automatically for structural models if no other estimator has been specified by the user. A structural model is identified by $F_0 \neq I_K$, i.e. if structural coefficients were specified in F_0.

Reduced form VEC model

Model (3.7) can be estimated in two different ways, the Johansen ML approach (Johansen (1995)) and a two stage estimation (see, e.g., Lütkepohl (1993a, Chapter 11)).

The Johansen ML approach:

Currently, only the sub model

$$\begin{aligned} \Delta y_t &= \overline{\alpha}(\beta y_{t-1} + \beta^{(d)} d_t^{(ec)}) \\ &\quad + \overline{F} Y_{t-1} + C^{(sys)} d_t^{(sys)} + u_t \end{aligned} \tag{3.14}$$

can be estimated by the Johansen approach. The estimators $\widehat{\overline{\alpha}}_{JOH}$, $\widehat{\overline{F}}_{JOH}$, $\widehat{\overline{C}}^{(sys)}_{JOH}$, and $[\widehat{\beta}_{JOH} : \widehat{\beta^{(d)}}_{JOH}]$ are described in Johansen (1995, Chapter 6), their

distribution in Johansen (1995, Chapter 13). Since $\widehat{\widehat{\alpha}}_{JOH}$ and $[\widehat{\beta}_{JOH} : \widehat{\beta^{(d)}}_{JOH}]$ are not unique, the cointegration vectors $[\widehat{\beta}_{JOH} : \widehat{\beta^{(d)}}_{JOH}]$ are normalized as in Johansen (1995, Chapter 13.2).

The two-step approach:

As the name suggests the procedure consists of two steps. The first step either estimates or imposes cointegration vectors $[\beta : \beta^{(d)}]$ and the second stage estimates the remaining parameters conditionally on the first stage.

As outlined above, r_1 cointegration vectors, $0 \leq r_1 \leq r$, are estimated from the data, and $r_2 = r - r_1$ cointegration vectors are imposed. If $r_1 = 1$ the cointegration relation can be estimated by OLS or by Johansen's ML approach. If $r_1 > 0$ only Johansen's ML approach is available for estimating the cointegration vectors. The remaining $r_2 > 0$ cointegration vectors must be specified by hand, see Figure 3.5.

The OLS and EGLS estimators of the VEC model (3.7) have the same functional form as for the VAR model (3.2). However, now they are based on the compact form (3.9), which defines Y, $\overline{G}$, and Z differently. The estimators are not repeated here, see page 76.

Selection of estimation method

Before estimating the VEC model the *first* time the user is forced to select *Estimation→Estimation Strategy* from the VEC sub window. An information about the current estimation strategy is given. The strategy can be changed from the two-step approach to Johansen's ML approach.

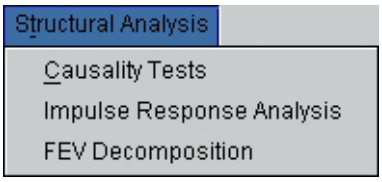
For the two step approach there are options for specifying the estimation method of the first step and second step. For the second step four different options are available: Automatic, OLS, EGLS, and 3SLS. Automatic chooses OLS, EGLS, or 3SLS estimation depending on the restrictions imposed.

Reformulation of the VEC model

Using the equivalence of the VEC form and the VAR form on page 69, the VEC model can be rewritten in the usual VAR form. Since some of the tools used later are based on the VAR form, the estimated VEC is automatically rewritten and additionally displayed in VAR form.

3.2.9 Structural analysis

Model (3.1) summarizes the instantaneous and intertemporal relations between the variables in the vector y. The exact form of these relations is usually difficult to see directly from the A_i coefficients. Tools for analyzing these relations are provided in the menu *Structural analysis* of the VAR and VEC sub windows.



The menu is the same for the VAR and VEC analyses. It contains entries for causality tests, impulse response analysis, and forecast error variance decomposition. Causality tests and forecast error variance decompositions are only sensible in the context of VAR(p) models of dimension $K > 1$ and order $p > 0$. In all other cases these two options are disabled. Furthermore, the model must be estimated before forecast error variance decomposition and impulse response analysis can be conducted.

The tools provided are standard in the VAR literature. However, inferential methods were refined in the past (Sims and Zha (1999), Benkwitz et al. (2000)). Especially the various bootstrap confidence intervals for impulse responses of full and subset VAR and VEC models offered by JMulTi are currently not found in any other statistical software package.

Causality tests

Tests for Granger non-causality and instantaneous causality can be performed when selecting *Structural Analysis→Causality Tests* from the VAR or VEC sub window. The concepts and tests are described, for example, in Lütkepohl (1993a) and Hamilton (1994). Also note the critique of instantaneous and Granger causality expressed there.

Test for Granger non-causality:
There are different possibilities to translate the concept of Granger-causality in the context of a VAR analysis and to develop a test based on a specific translation.

JMulTi translates Granger-causality as shown in Lütkepohl (1993a, Chapter 2): Granger-causality can be expressed with the coefficients of the A_i parameter matrices. For that, the K variables in y_t are classified into a $K_{\mathbf{1}}$-dimensional vector $y_{\mathbf{1},t}$ and a $K_{\mathbf{2}}$-dimensional vector $y_{\mathbf{2},t}$ ($K = K_{\mathbf{1}} + K_{\mathbf{2}}$)

$$y_t = \begin{bmatrix} y_{\mathbf{1},t} \\ y_{\mathbf{2},t} \end{bmatrix}.$$

The model parameter matrices of (3.1) are accordingly written in a block structure

$$A_i = \begin{bmatrix} A_{\mathbf{11},i} & A_{\mathbf{12},i} \\ A_{\mathbf{21},i} & A_{\mathbf{22},i} \end{bmatrix}$$

with $A_{\boldsymbol{kl},i}$ having dimension $(K_{\boldsymbol{k}} \times K_{\boldsymbol{l}})$, $\boldsymbol{k} = \mathbf{1}, \mathbf{2}$ and $\boldsymbol{l} = \mathbf{1}, \mathbf{2}$. The process $y_{\boldsymbol{k},t}$ is causal for $y_{\boldsymbol{l},t}$ in Granger's sense if there exists at least one $A_{\boldsymbol{lk},i} \neq 0$, $\boldsymbol{l} \neq \boldsymbol{k}$.

The respective blocks in the parameter matrices can be tested whether they jointly differ significantly from zero. This will create a test for Granger non-causality. The statement that $y_{\boldsymbol{k},t}$ does not Granger-cause $y_{\boldsymbol{l},t}$ translates into $A_{\boldsymbol{lk},i} = 0$ for $i = 1 \ldots, p$, $\boldsymbol{k} \neq \boldsymbol{l}$. The test must check this parametric expression. In the cointegration analysis the test procedure fits the VAR-form

of the VEC model. However, it does not estimate a VAR(p) but a VAR($p+1$) model and completely ignores A_{p+1} in the test statistic.

The Wald test of Lütkepohl (1993a) is used here. A Wald statistic is based on the unconstrained estimator which is asymptotically normal. That is, an unrestricted VAR (3.2) is estimated first and then tested whether the coefficients in $A_{\boldsymbol{kl},i}$ are jointly not significantly different from zero:

$$\begin{aligned} \mathrm{H}_0: \quad S\overline{g} &= 0 \\ \mathrm{H}_1: \quad S\overline{g} &\neq 0 \end{aligned}$$

where S is defined as in (3.11). The test statistic

$$\lambda_w = (S\overline{g})' \left(S\widehat{\Sigma}_{\widehat{g}_{OLS}} S' \right)^{-1} (S\overline{g}) \tag{3.15}$$

is computed and adjusted as follows:

$$\lambda_F = \lambda_w / (pK_{\boldsymbol{k}}K_{\boldsymbol{l}})$$

which takes into account the use of the estimator $\widehat{\Sigma}_{\overline{g}}$. The statistic λ_F is reported and should behave like a $F(pK_{\boldsymbol{k}}K_{\boldsymbol{l}}, T-N)$ under H_0. Granger non-causality can be rejected at significance level γ if $F_{pK_{\boldsymbol{k}}K_{\boldsymbol{l}},T-N}(x < \lambda_f) < 1-\gamma$.

Test for instantaneous causality:

A condition for instantaneous causality is given by

$$\mathrm{E}(u_{\mathbf{1}t} u'_{\mathbf{2}t}) = 0, \tag{3.16}$$

see Lütkepohl (1993a). This condition translates to zero constraints for $\boldsymbol{\sigma} = \mathrm{vech}(\Sigma_u)$, where the vech operator stacks all elements on and below the main diagonal of Σ_u. $\widehat{\Sigma}_u$ is estimated from the unrestricted VAR model (3.2). Next it is tested whether the identified coefficients of $\boldsymbol{\sigma}$ in (3.16) are jointly not significantly different from zero:

$$\begin{aligned} \mathrm{H}_0: \quad S\boldsymbol{\sigma} &= 0 \\ \mathrm{H}_1: \quad S\boldsymbol{\sigma} &\neq 0 \end{aligned}$$

JMulTi computes the test statistic

$$\lambda_w = T\widetilde{\boldsymbol{\sigma}}' S' \left(2S\boldsymbol{D}_K^+(\widetilde{\Sigma}_{\widehat{u}_{OLS}} \otimes \widetilde{\Sigma}_{\widehat{u}_{OLS}})\boldsymbol{D}_K^{+\prime} S'\right)^{-1} S\widetilde{\boldsymbol{\sigma}},$$

where $\boldsymbol{D}_K^{+'}$ is the Moore-Penrose inverse of the duplication matrix $\boldsymbol{D}_K$, and S is a $(v \times K(K+1)/2)$ matrix of rank v, see Lütkepohl (1993a, Chapter 3). Under H_0, λ_w is asymptotically $\chi^2(v)$ distributed.

Impulse response analysis

Impulse responses are often used for interpreting the relation between the endogenous variables modeled in a VAR. There are different impulse responses depending on the kind of impulse hitting the system. For detailed discussions see Sims (1980), Lütkepohl (1993a), and Lütkepohl and Breitung (1997)). JMulTi computes so-called forecast error and orthogonal impulse responses.

Impulse responses considered:

The forecast error impulse response (see Lütkepohl (1991, Sec. 2.3.2)), $\phi_{ij,h}$ traces out the expected response of $y_{i,t+h}$ to a unit change in $y_{j,t}$ holding constant all past values of y_t, $Y_{-1} = \{y_{t-1}, y_{t-2}, \ldots\}$. The impulse response $\phi_{ij,h}$ is the ijth element of the matrix Φ_h obtained as

$$\Phi_h = A_0^{-1} \sum_{i=1}^{h} \Phi_{h-i} A_i, \quad h = 1, 2, \ldots, \tag{3.17}$$

where $\Phi_0 = A_0^{-1}$, and $A_i = 0$ for $i > p$. Since the change in y_{it} given Y_{-1} is measured by the innovation u_{it} and since the u_t are the 1-step ahead forecast errors Φ_h are called forecast error impulse responses (Lütkepohl (1993a)). The estimator of Φ_h is

$$\widehat{\Phi}_h = \widehat{A}_0^{-1} \sum_{i=1}^{h} \widehat{\Phi}_{h-i} \widehat{A}_i, \quad h = 1, 2, \ldots, \tag{3.18}$$

with $\widehat{\Phi}_0 = I_K$, and $\widehat{A}_i = 0$ for $i > p$.

Tracing out the effects of shocks that are not isolated in one variable but reflect the correlation structure of the error terms in u_t lead to the so-called orthogonal impulse responses

$$\Theta_h = \Phi_h P, \qquad A_0 = I_K \tag{3.19}$$

where Φ_h is defined as in (3.17) and where P is the lower triangular Choleski decomposition of $\not\Sigma_u$ such that $\not\Sigma_u = PP'$ (Lütkepohl (1993a)). The condition $A_0 = I_K$ indicates that this type of impulse response is only computed for reduced form models. Since Θ_h can be derived from a transformed model of (3.2) with diagonal residual covariance matrix (i.e. the transformed residuals do not have contemporaneous correlation, are orthogonal) this type of impulse response is called orthogonal. Θ_h is estimated, of course, by

$$\widehat{\Theta}_h = \widehat{\Phi}_h \widehat{P}. \tag{3.20}$$

Inference:

In order to measure estimation uncertainty JMulTi computes three different bootstrap confidence intervals. Bootstrap confidence intervals enjoy much popularity. They are regarded as being more reliable than confidence intervals based on first order asymptotic theory. This view is discussed extensively later in Chapter 4.

The following residual based bootstrap method is considered:

1. Estimate the parameters of the model (3.1) by a suitable procedure.

2. Generate bootstrap residuals $u_1^*, \ldots, u_T^*$ by randomly drawing with replacement from the set of estimated and re-centered residuals, $\{\widehat{u}_1 - \bar{u}_\cdot, \ldots, \widehat{u}_T - \bar{u}_\cdot\}$, where $\widehat{u}_t$ is defined as in (3.12), and $\bar{u}_\cdot = T^{-1} \sum \widehat{u}_t$.

3. Set $(y_{-p+1}^*, \ldots, y_0^*) = (y_{-p+1}, \ldots, y_0)$ and construct bootstrap time series recursively using the levels representation given in (3.1),

$$y_t^* = \widehat{A}_0^{-1} \left(\widehat{A}_1 y_{t-1}^* + \cdots + \widehat{A}_p y_{t-p}^* + \widehat{B} X_t + \widehat{C} D_t + u_t^* \right), \quad t = 1, \ldots, T.$$

4. Reestimate the model parameters using y_t^* instead of y_t.

5. Calculate a bootstrap version of the statistic of interest, say $\widehat{\phi}_{ij,h}^*$ and $\widehat{\theta}_{ij,h}^*$, based on the parameter estimates obtained in Stage 4.

Note that Stage 4 applies the same estimation procedure of Stage 1. This is important in the cointegration analysis. The cointegration vector is reestimated in every bootstrap replication if it was estimated in Stage 1. Otherwise all cointegration vectors are held constant during the bootstrap if they were parametrically imposed to the model before estimation. In other words, the bootstrap algorithm respects the information on modeling the cointegration relation as shown in Figure 3.5.

In the following the symbols ϕ, $\widehat{\phi}_T$ and $\widehat{\phi}_T^*$ denote some general impulse response coefficient $\phi_{ij,h}$ or $\theta_{ij,h}$, its estimator implied by the estimators of the model coefficients and the corresponding bootstrap estimator, respectively. The subscript T indicates the sample size. Furthermore, denote $\boldsymbol{Y} = \{y_{-p+1}, \ldots, y_0, \ldots, y_T\}$ and $\boldsymbol{X} = \{x_{-q+1}, \ldots, x_0, \ldots, x_T\}$.

The most commonly used method in setting up confidence intervals for impulse responses in practice proceeds by using $\gamma/2$- and $(1-\gamma/2)$-quantiles, say $s_{\gamma/2}^*$ and $s_{(1-\gamma/2)}^*$, respectively, of the bootstrap distribution $\mathcal{L}(\hat{\phi}_T^* \mid \boldsymbol{Y}, \boldsymbol{X})$, and defining

$$CI_S = \left[s_{\gamma/2}^*, s_{(1-\gamma/2)}^*\right]. \tag{3.21}$$

The interval CI_S is the percentile confidence interval described, e.g., by Efron and Tibshirani (1993).

Other intervals proposed in the bootstrap literature (see, e.g., Hall (1992)) are also available in JMulTi. Let $t_{\gamma/2}^*$ and $t_{(1-\gamma/2)}^*$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of

$$\mathcal{L}(\hat{\phi}_T^* - \hat{\phi}_T \mid \boldsymbol{Y}, \boldsymbol{X}),$$

respectively. According to the usual bootstrap analogy,

$$\mathcal{L}(\hat{\phi}_T - \phi) \approx \mathcal{L}(\hat{\phi}_T^* - \hat{\phi}_T \mid \boldsymbol{Y}, \boldsymbol{X}),$$

one gets the interval

$$CI_H = \left[\hat{\phi}_T - t^*_{(1-\gamma/2)}, \hat{\phi}_T - t^*_{\gamma/2}\right]. \tag{3.22}$$

Hall (1992, page 85) calls CI_H "percentile interval" as well, and denotes on page 12 CI_S as the "other percentile method". Therefore, in JMulTi the method leading to CI_H is referred to as *Hall's percentile CI*, whereas the method underlying CI_S is referred to as *Efron's percentile CI*.

A studentized version of CI_H is also available. It uses the statistic $(\hat{\phi}_T - \phi)/\sqrt{\widehat{\text{var}}(\hat{\phi}_T)}$ as a basis for constructing confidence intervals. Hence in the present context it may be useful to determine a bootstrap quantile based on the statistic $(\hat{\phi}_T^* - \hat{\phi}_T)/\sqrt{\widehat{\text{var}}(\hat{\phi}_T^*)}$. In this approach the variances are also estimated by a bootstrap, that is,

$$\widehat{\text{var}}(\hat{\phi}_T) = \frac{1}{B^* - 1} \sum_{i=1}^{B^*} \left(\hat{\phi}_T^{*,i} - \overline{\hat{\phi}_T^*}\right)^2$$

and

$$\widehat{\text{var}}(\hat{\phi}_T^*) = \frac{1}{B^{**} - 1} \sum_{i=1}^{B^{**}} \left(\hat{\phi}_T^{**,i} - \overline{\hat{\phi}_T^{**}}\right)^2,$$

where $\hat{\phi}_T^{**,i}$ is obtained by a double bootstrap, that is, pseudo-data are generated according to a process obtained on the basis of the bootstrap systems parameters and B^* and B^{**} are the respective numbers of bootstrap replications in the first and second stages (see Hall (1992) for details).

Let $t^{**}_{\gamma/2}$ and $t^{**}_{(1-\gamma/2)}$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles, respectively, of

$$\mathcal{L}\left((\hat{\phi}_T^* - \hat{\phi}_T)/\sqrt{\widehat{\text{var}}(\hat{\phi}_T^*)} \,\middle|\, \boldsymbol{Y}, \boldsymbol{X}\right).$$

Using these quantiles the studentized Hall interval is

$$CI_{SH} = \left[\hat{\phi}_T - t^{**}_{(1-\gamma/2)}\sqrt{\widehat{\text{var}}(\hat{\phi}_T)},\ \hat{\phi}_T - t^{**}_{\gamma/2}\sqrt{\widehat{\text{var}}(\hat{\phi}_T)}\right]. \tag{3.23}$$

The question arises which interval should be used in addition to the estimated impulse response. Chapter 4 discusses the intervals CI_S, CI_H, CI_{SH} and alternative bootstrap confidence intervals in detail. A Monte-Carlo study shows the small sample properties of these intervals in cases of special interest in empirical work. See also the next section in this chapter. It demonstrates the bootstrap confidence intervals contained in JMulTi in the impulse response analysis of a German monetary system.

Forecast error variance decomposition

The difference between the realization y_{t+h} and its h-step forecast $y_t(h)$ is called forecast error. The forecast error variance decomposition $\omega_{jk,h}$ analyzes the contribution of innovations in variable k to the forecast error variance of the h-step forecast of variable j.

In JMulTi the they are estimated as

$$\widehat{\omega}_{jk,h} = \sum_{i=0}^{h-1} \widehat{\theta}_{jk,i}^2 / \widehat{\mathrm{MSE}}[y_{j,t}(h)]$$

where $\widehat{\theta}_{jk,i}$ is the (j,k)-th element of $\widehat{\Theta}_i$ in (3.20) and the MSEs of the $y_{j,t}$ variables are taken from the diagonal elements of

$$\widehat{\mathrm{MSE}}[y_t(h)] = \sum_{i=0}^{h-1} \widehat{\Theta}_i \widehat{\Theta}_i'.$$

Chapter 4 discusses problems related to asymptotic confidence intervals for impulse responses. There are cases where these intervals are problematic. Similar problems exist for forecast error variance decompositions, for instance if they are 0. Consequently, confidence intervals for $\widehat{\omega}_{jk,h}$ are currently not estimated in JMulTi.

3.3 Analyzing a German monetary system with JMulTi

The model for the German monetary sector as analyzed by Brüggemann and Wolters (1998) will be considered next. Their work investigates channels of German monetary policy. For that purpose, a subset VEC model is specified and estimated. It is interpreted with an impulse response analysis. However, no measure is reported for assessing the estimation uncertainty of impulse responses. This is provided in Benkwitz et al. (2001). Their paper computes and compares various bootstrap confidence intervals for the M1 system of Brüggemann and Wolters (1998) and the M3 system of Lütkepohl and Wolters (1998).

This section demonstrates how JMulTi can be used to analyze the M1 system. In addition the roubustness of the results of Brüggemann and Wolters (1998) is analyzed by using alternative VAR models. Another question is addressed in the last subsection. It is related to differences of various bootstrap confidence intervals in empirical analyses. Monte-Carlo studies often recommend to apply a specific method in the empirical analysis. For example, Chapter 4 recommends to use CI_H. Nevertheless, it is interesting to see to what extend the most often used bootstrap confidence intervals differ when they are computed for impulse responses of an empirical VEC model. Comparative results are reported for an unconstrained reduced form VAR model in Benkwitz et al. (2001). In contrast to that, different bootstrap confidence intervals for the VEC model as specified by Brüggemann and Wolters (1998) are compared in this section.

3.3.1 Loading data set

The data set is provided in the file `BW_M1.DAT`. It is available from `http://www.jmulti.de/download.html` and can be loaded into the program with the load tool (see Subsection 3.1.5 for details). The data set contains quarterly seasonally unadjusted data. The following variables are included: $M1$ is nominal M1; GNP is the real GNP; P is the GNP deflator; R is a long-term interest rate ('Umlaufsrendite'); PM is an import price index. In addition there are a number of deterministic variables in the data set such as seasonal dummies and a shift dummy $S90Q3$ which takes into account the level shifts in $M1$ and GNP due to the German re-unification. It equals zero until 1990(2) and afterwards it has the value one. Data sources are documented in Brüggemann and Wolters (1998) and are reproduced in the comment section of the data file.

3.3.2 Initial analysis

Brüggemann and Wolters (1998) construct quarterly models for the period 1962(1) - 1989(4) and the extended period 1962(1) - 1996(2). In the following the model version for the extended period 1962(1) - 1996(2) is analyzed. It includes German re-unification in 1990. The system comprises the following variables: $m1_t$ is the logarithm of $M1$; y_t is the logarithm of GNP; p_t is the logarithm of P, hence, $(m1 - p)_t$ is the logarithm of real M1 and $\Delta p_t = p_t - p_{t-1}$ is the quarterly inflation rate; R_t is R, the long-term interest rate ('Umlaufsrendite'); pm_t is the import price index PM which is treated as an unmodelled variable reflecting the openness of the German economy and capturing the effects of exchange rates. In addition the model contains a number of deterministic variables such as a constant term, seasonal dummies and the shift dummy $S90Q3$ which captures the effect of German re-unification.

The transformation tools in the Initial Analysis can be used for transforming

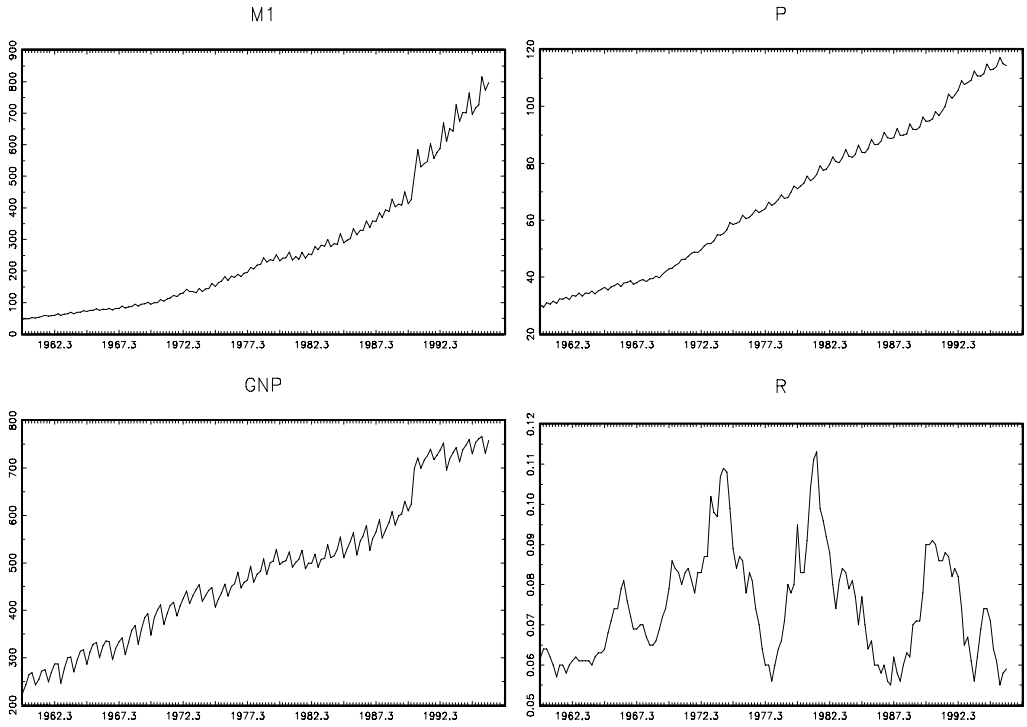


Figure 3.8: Plot of nominal M1, prices (P), real GNP and long-term interest rate (R).

the original series. It is also possible to load the variables directly from the data file `BW_M1_TRANS.dat` which includes the transformed series.

With the plot tool which is found in the Initial Analysis sub frame plots of the untransformed series M1, P, GNP, and R are created. These plots are displayed in a graphic window. The window has a menu with functions for zooming, converting, and printing. Thus all plots can be saved and included in documents later. For instance, Figure 3.8 is a "JMulTi" graphic. It is created from a zoomed GAUSS graphic (to cut off the main title) which was converted (i.e. exported) and imported in this document.

Brüggemann and Wolters (1998) found evidence that the variables $m1_t$, p_t, y_t and R_t are $I(1)$ and cointegrated with cointegration rank $r = 1$. These results can easily be reproduced using the UR Tests and Cointegration Test tools

of the Initial Analysis. For the period from 1961(4) to 1996(2) Brüggemann and Wolters (1998, Equation (3.4)) found the following long-run money demand relation

$$(m1 - p)_t = 1.105y_t - 5.133R_t + 0.407S90q3_t + ec_t. \tag{3.24}$$

Here ec_t stands for the deviations from the long-run relation.

3.3.3 Model specification

Brüggemann and Wolters (1998) estimate the VEC model (3.5) by imposing the long-run money demand relation (3.24). The estimation results of JMulTi are given in Table 3.1. There are small differences compared to the results reported in Brüggemann and Wolters (1998, Table 4.6) due to different estimation methods (in JMulTi GLS vs. 3SLS) and different statistical software (in JMulTi GAUSS vs. EViews2.0).

This model is the result of a specification procedure described in detail in Brüggemann and Wolters (1998). Lagged differences with insignificant coefficients are removed step by step starting with a specification which contains differenced values up to the fourth lag. Regardless of the t-values of the coefficients of the error correction term, this term is included in the regressions until all insignificant lagged differences have been eliminated. The error correction term is only eliminated if it turns out to be insignificant in the model in which the other insignificant terms had already been eliminated. The deterministic variables are not eliminated during the specification procedure even if the t-value of the respective coefficient indicated an insignificant value.

The specification procedure can be reproduced in JMulTi with the search algorithm for subset constraints. Since the algorithm imposes slightly different restrictions the zero constraints of Brüggemann and Wolters (1998) have been entered by hand here.

Table 3.1: Estimation results of M1 system. Note that every equation contains all deterministic variables. Their coefficients are not shown here. Estimation Period is 1962(1) - 1996(2)

	$\Delta m1_t$	Δp_t	Δy_t	ΔR_t
ec_{t-1}	-0.116 (-6.688)[a]			
$\Delta m1_{t-1}$				0.045 (2.496)
$\Delta m1_{t-2}$			0.195 (3.042)	
$\Delta m1_{t-3}$			0.112 (1.967)	
Δp_{t-1}		-0.129 (-2.500)		
Δp_{t-2}		0.203 (3.708)	-0.286 (-3.042)	
Δp_{t-4}	0.411 (5.385)	0.541 (10.245)		
Δy_{t-2}	-0.226 (-4.772)	0.085 (3.759)	-0.436 (-6.497)	
Δy_{t-3}		0.118 (5.631)		
Δy_{t-4}			0.455 (7.551)	
ΔR_{t-1}	-0.877 (-3.627)	0.247 (2.440)		0.187 (2.277)
Δpm_{t-4}		0.065 (2.757)		

[a] Absolute values of t-ratios in parentheses.

Table 3.2: Residual correlation matrix of M1 System

	$m1_t$	p_t	y_t	R_t
$m1_t$	1			
p_t	0.180	1		
y_t	0.170	-0.078	1	
R_t	-0.161	0.001	0.171	1

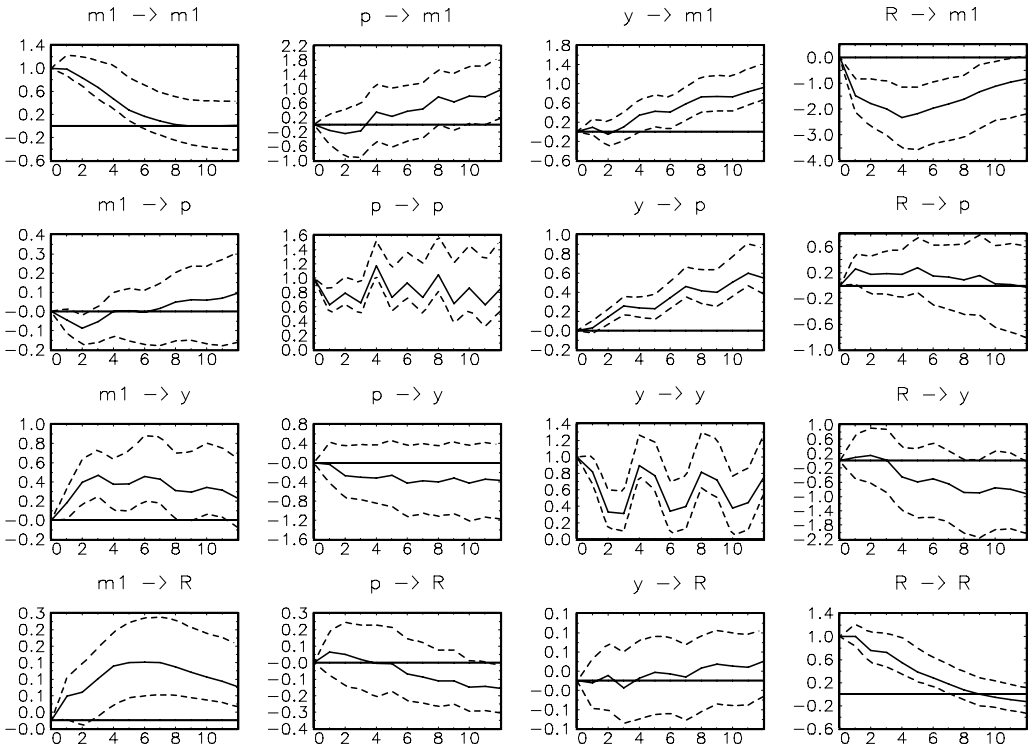


Figure 3.9: Estimated impulse responses for full VAR(5) model with 95% CI_H.

Since the model is estimated in reduced form (3.7), a fully unrestricted version with full rank error correction term can be estimated by considering the VAR in (3.2) with order $p = 5$. This is done in the next subsection. Table 3.2 shows quite small instantaneous residual correlation. Therefore no orthogonalization is needed for computing meaningful impulse responses.

3.3.4 VAR(5) model

Figure 3.9 shows the estimated impulse responses of a full VAR(5) model with 95% Hall percentile bootstrap confidence intervals (CI_H). The intervals for this and the following model versions were computed using $B = 1,000$ bootstrap replications. The sensitivity of the results with respect to the number of bootstrap replications is easy to check in JMulTi. Every computed confidence interval is stored in the program and listed with a label that also includes B. It

turns out that in this case similar intervals are obtained even if only $B = 100$ replications are used.

A major problem with the intervals is that they are rather wide and, hence, the actual responses in the underlying system are quite uncertain if the intervals properly reflect the estimation variability. For example, based on the confidence intervals in Figure 3.9, an impulse in $m1$ does not have a significant effect on the price level (`m1` $\rightarrow$ `p`). Moreover, an increase in the price level does not have a significant impact on income (`p` $\rightarrow$ `y`). Thus, an impulse response analysis based on an unrestricted reduced form model does not give a clear indication of the relations between the variables. The results in Figure 3.9 also show the importance of computing CIs for the impulse responses because an interpretation that ignores the substantial estimation uncertainty may be misleading.

3.3.5 The full VEC model

In a next step the cointegration vector is fixed to Equation (3.24) and the VEC model (3.5) is estimated. In Figure 3.10 the estimated impulse responses and CI_H intervals are shown. In order to see the effects of imposing the cointegration restriction, intervals of the full VAR are also displayed. The cointegration relation is not reestimated in every bootstrap replication. See Benkwitz et al. (2001) for a discussion about the effect of fixing the cointegration relation in contrast to a reestimation in every bootstrap replication.

In nearly all cases both models show similar confidence intervals up to the forcast horizon 4. The deviation for higher forecasts is sometimes considerable, for example `m1` $\rightarrow$ `m1` or `R` $\rightarrow$ `R`. The estimated impulse responses and their confidence intervals in the VEC model reflect in these cases the parametric constraints associated with the VEC model. Interestingly enough, the size of the intervals is in most cases the same. But there are also cases where the es-

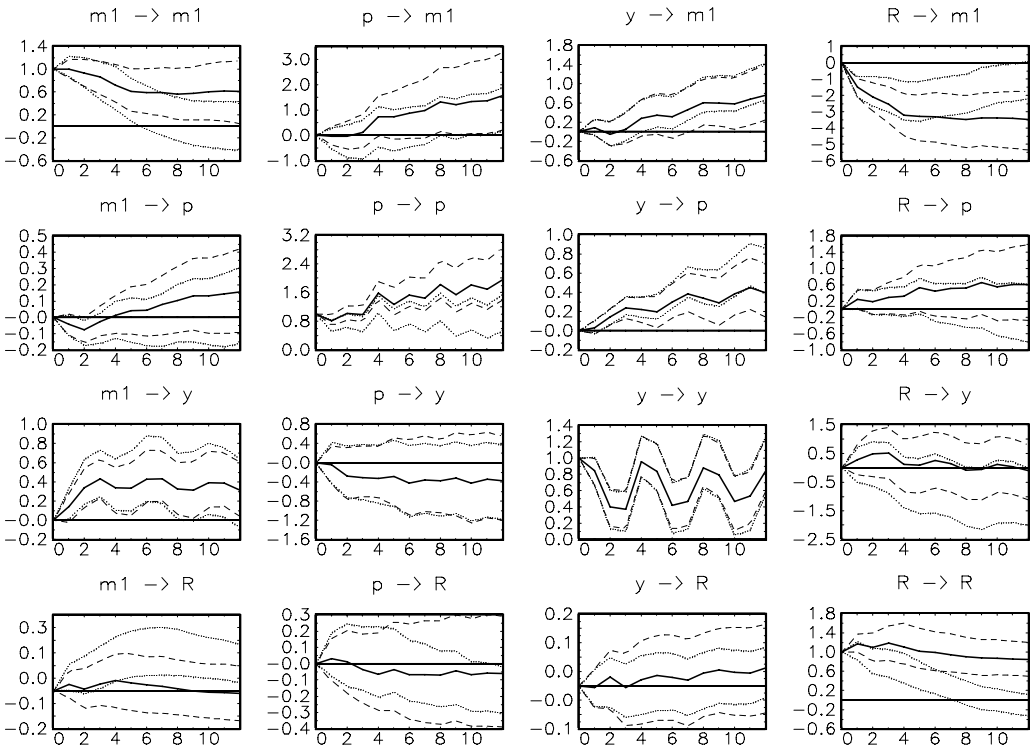


Figure 3.10: Estimated impulse responses for full VEC model with 95% CI_H (dashed lines) and 95% CI_H from the full VAR (dotted lines).

timation uncertainty seems to increase when estimating the impulse responses from the VEC model. It is also possible that the intervals from the VAR model are distorted due to missing constraints (Benkwitz et al. (2001)).

3.3.6 The subset VEC model

An improvement in the estimation precision can be expected from taking into account the restrictions imposed by Brüggemann and Wolters (1998). The results are shown in Figure 3.11. The impulse responses are computed from the subset VEC model as presented in Table 3.1.

As for the full VEC model the cointegration vector is fixed in each bootstrap replication. The CI_H intervals from the full VEC model are included for comparison purposes. The comparison indicates that the precision of the

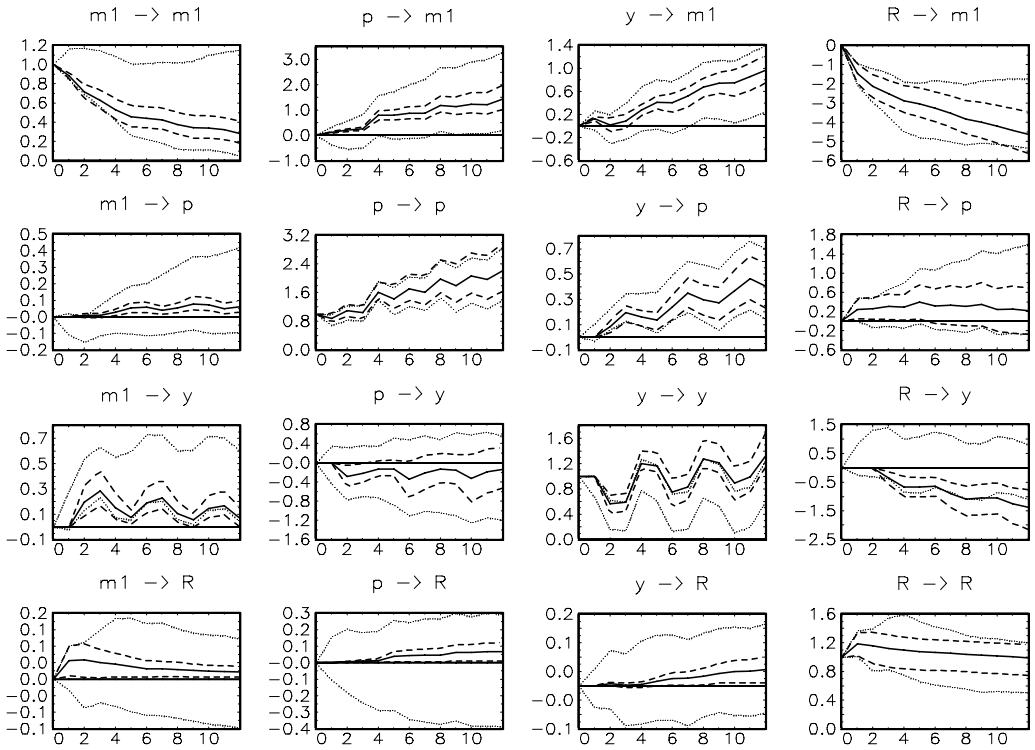


Figure 3.11: Estimated impulse responses for subset VEC model with pointwise 95% CI_H (dashed lines) and 95% CI_H from the full VEC (dotted lines).

estimated impulse responses from the subset VEC model increased. Now the response of $m1$ to an impulse in the price level p has become significant and the same is true for the response of p to an impulse in $m1$, for instance. Thus, the interpretation from Brüggemann and Wolters (1998) that the impact of changes in $m1$ on the price level may not be very strong is not confirmed here.

In Figure 3.11 it is interesting to see that the impulse responses from the subset model are in most cases within the intervals from the unrestricted model, in particular for low lags. On the other hand, the intervals from the subset model do not always contain the estimates of the impulse responses from the unrestricted model. Hence, estimating the impulse responses from an unrestricted model does not only increase the uncertainty of the estimates but may also lead to quite different point estimates. The long-run development of the

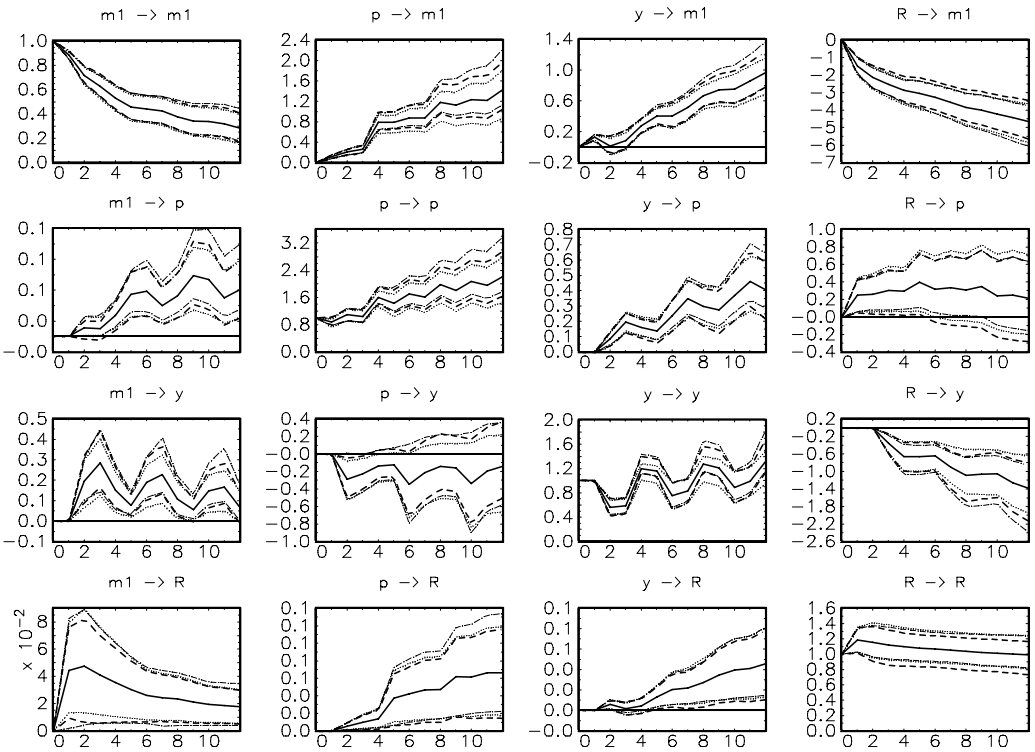


Figure 3.12: Comparison of bootstrap confidence intervals for impulse responses computed from the subset VEC model. The plot shows pointwise 95% confidence intervals: CI_H (strong dashes), CI_S (fine dots), and CI_{SH} (fine dots and dashes)

impulse responses from both models is similar due to enforcing the cointegration restriction.

It may also be worth noting that using the bootstrap for an unrestricted model may result in singularities in the asymptotic distributions of the estimated impulse responses. This in turn may lead to strongly distorted and, hence, unreliable bootstrap confidence intervals as pointed out in Chapter 4. Thus, using a subset model is also useful for removing one source of problems for the bootstrap confidence intervals.

3.3.7 Comparison of bootstrap confidence intervals

A comparison of the three bootstrap confidence intervals in JMulTi is carried out next. It discusses the question to what extend the confidence intervals differ in a real empirical analysis. For that, the intervals CI_H, CI_S, and CI_{SH} were computed. The results are shown in Figure 3.12.

For the confidence intervals based on a studentized statistic (CI_{SH}) 50 bootstrap drawings for estimating $\widehat{\text{var}}(\hat{\phi}_T^*)$ were used. Furthermore, the bootstrap algorithm on page 84 uses the same seed for pseudo randomly drawing the residuals in Step 2 when computing the intervals CI_H/ CI_S and CI_{SH}. Therefore, the intervals are computed from the same bootstrap time series. The differences in the intervals in Figure 3.12 are completely due to differences between the methods.

Clearly in this case the differences between the methods are not substantial. Because in most cases the confidence intervals are almost symmetric around the estimated impulse response coefficients it is not surprising that CI_S and CI_H are similar. Also, the CI_{SH} intervals are quite similar to CI_H in most cases. The interpretation of the estimated impulse response does not depend on the type of confidence interval used in this example. However, this statement cannot be generalized. There are examples where the choice of the type of the confidence interval affects the interpretation (e.g. Brüggemann and Lütkepohl (2001), Ehrmann et al. (2001)). Therefore, it is interesting to study these intervals in more detail. This will be done in the next chapter.

3.4 Summary

In this chapter the software JMulTi was introduced. The part of JMulTi which has been provided by the author was documented and discussed. The use in the VAR-framework of JMulTi has been demonstrated on the basis of a small

monetary system for Germany.

JMulTi makes methods from most recent research available to empirical analysis. For example, the impulse response analysis provides various bootstrap confidence intervals which are not available in other popular software packages for dynamic econometric analysis. They either do not provide confidence intervals for impulse responses (e.g. PcFiml) or they do provide them but only for simple unrestricted VARs (e.g. EViews)

JMulTi is easy and intuitively to use because it is a fully menu-driven software. Therefore, it is a good learning and teaching software. Until now, JMulTi has been used in econometric courses at the Humboldt University, Berlin and at the European University Institute, Florence.

It has turned out that the capabilities of JMulTi in the context of the VAR models are good for a "general to specific" modeling strategy. Many useful tools help to specify and estimate the models fast and reliable. The specification of zero restrictions in the model coefficient matrices is a good example. A simple mouse click on a matrix element includes or excludes the respective variable from the restricted regression. It is also possible to use an algorithm to search for zero constraints.

The last section finished with an interesting result. The compared bootstrap confidence intervals do not show differences that are important for the interpretation. However, this statement cannot be generalized. The shown intervals are symmetric around the estimated impulse response. In this case differences between CI_H and CI_S cannot be expected. But there are studies (e.g. Brüggemann and Lütkepohl (2001), Ehrmann et al. (2001)) with extremely unsymmetric intervals. In such cases it matters what interval (CI_H or CI_S) has been used. Nevertheless, the small differences between CI_H and CI_{SH} are astonishing and question the computational burden that is connected to CI_{SH}.

Chapter 4

Bootstrap confidence intervals for impulse responses

In Section 3.3 of the previous chapter it was shown how VAR models can be used for analyzing and understanding economic systems. An impulse response analysis has been conducted and various bootstrap confidence intervals were computed. Two known problems related to the inference in the impulse response analysis were mentioned. They can arise when computing confidence intervals for estimated impulse responses using the first order asymptotic method (see Lütkepohl (1993a)).

This chapter analyzes the question whether the known problems can be solved by using bootstrap methods (see Efron (1979)). The work presented here is based on Benkwitz et al. (2000) and Benkwitz et al. (2001).

In fact, bootstrap confidence intervals (CIs) or regions are often reported in empirical studies because they are regarded as being more reliable than confidence intervals based on asymptotic theory. Support for this view seemingly comes from the skewness of the bootstrap intervals which contrasts with the symmetry of standard asymptotic intervals. Some Monte Carlo studies have confirmed this belief (see, e.g., Fachin and Bravetti (1996), Kilian (1998b)).

On the other hand, it was also found that in some cases bootstrap CIs are not very reliable. There are studies showing that they can lead to extremely poor CIs with actual confidence content substantially different from the nominal level (e.g., Griffiths and Lütkepohl (1989), Fachin and Bravetti (1996), Kilian (1998a), Kilian (1998b)). Of course, this may partly be a small sample problem and hence small sample modifications and corrections have been proposed (e.g., Kilian (1998a), Kilian (1998b)). Although these modifications are quite successful in some cases, it will turn out in the simulations that they do not help to solve the particular problems encountered with impulse responses in certain regions of the parameter space.

This chapter will show that in addition to these small sample problems there are also conceptual problems that prevent the usual asymptotic and bootstrap CIs for impulse responses from having the correct probability content even asymptotically. Sims and Zha (1999) also launched a critique of the usual approaches to construct CIs for impulse responses. Their critique is based on a Bayesian point of view, however. In contrast, the following treatment will remain within the classical asymptotic framework. It is argued that even with this treatment problems may arise.

The main problems result from the fact that the convergence rate of the estimators to their asymptotic distribution must remain constant over the whole parameter space. Otherwise, the standard asymptotics and the bootstrap will not work in the case considered here. It was noted, e.g., by Lütkepohl (1993a, Sec. 3.7), that this condition is not even satisfied for stationary VARs, let alone nonstationary ones. In particular, it is not satisfied for some cases of interest from an applied point of view. The problem is discussed in detail for the simplest case of a stationary univariate AR process of order one (AR(1)). It is clear that a method which has problems even in the simplest case cannot be expected to perform well for more complicated multivariate processes.

Therefore, the results question the precision of inference procedures which are in common use in applied macroeconomics. For the AR(1) case possible alternatives are considered and their potential for being generalized to higher order and to higher dimensional processes is discussed. Unfortunately, it turns out, that the methods which perform best in the simplest case are not easily generalizable.

A general framework of estimating impulse responses was presented in the last chapter. Inference on impulse responses is considered in Section 4.1 where also attention is drawn to some basic problems of asymptotic inference in the present context. A detailed analysis of the AR(1) case is provided and possible solutions are offered for this special case. A Monte-Carlo experiment is discussed in Section 4.3. Generalizations for more general cases and recommendations are discussed in Section 4.4.

The following notation is used in this chapter. The operator $o_P(\cdot)$ is the usual symbol for the order of convergence in probability. Furthermore, $\xrightarrow{d}$ signifies convergence in distribution and $\mathcal{L}(X)$ denotes the distribution of the random variable X, while F_X is used for the cumulative distribution function. $P(\cdot)$ is used to denote the probability of some event and $P_\alpha(\cdot)$ is used if the probability corresponding to a specific parameter α of the underlying distribution is of interest.

4.1 Inference on estimated impulse responses

Usually the coefficients of the model (3.1) are estimated by some standard procedure such as LS and estimators of the impulse responses are then obtained as

$$\hat{\phi}_{ij,h} = \phi_{ij,h}(\hat{A}_0, \hat{A}_1, \ldots, \hat{A}_p) \tag{4.1}$$

where the $\hat{A}_0, \ldots, \hat{A}_p$ are the estimated VAR coefficient matrices.

4.1.1 Standard asymptotic inference

Assuming that the $\hat{A}_i$ have an asymptotic normal distribution,

$$\sqrt{T}\text{vec}([\hat{A}_0, \ldots, \hat{A}_p] - [A_0, \ldots, A_p]) \xrightarrow{d} N(0, \Sigma_{\hat{A}}), \tag{4.2}$$

we have that the $\phi_{ij,h}$ have an asymptotic normal distribution as well,

$$\sqrt{T}(\hat{\phi}_{ij,h} - \phi_{ij,h}) \xrightarrow{d} N(0, \sigma^2_{ij,h}), \tag{4.3}$$

where

$$\sigma^2_{ij,h} = \frac{\partial \phi_{ij,h}}{\partial \alpha'} \Sigma_{\hat{A}} \frac{\partial \phi_{ij,h}}{\partial \alpha} \tag{4.4}$$

with $\alpha = \text{vec}[A_0, \ldots, A_p]$, and $\partial \phi_{ij,h}/\partial \alpha$ denotes a vector of partial derivatives. The result (4.3) holds if $\sigma^2_{ij,h}$ is nonzero which is a crucial condition for asymptotic inference to work. Note that $\Sigma_{\hat{A}}$ may be singular if there are constraints on the coefficients or if the variables are integrated and/or cointegrated (see Lütkepohl (1993a, Chapter 11)). However, even if $\Sigma_{\hat{A}}$ is nonsingular, $\sigma^2_{ij,h}$ may be zero because the partial derivatives in (4.4) may be zero. In fact, they will usually be zero in parts of the parameter space because the $\phi_{ij,h}$ generally consist of sums of products of the VAR coefficients and, hence, the partial derivatives will also be sums of products of such coefficients which may be zero.

To see the problem more clearly, consider the simple case of a one-dimensional AR(1) process $y_t = \alpha y_{t-1} + u_t$. In this case $\phi_h = \alpha^h$. Suppose $\hat{\alpha}$ is an estimator of α satisfying

$$\sqrt{T}(\hat{\alpha} - \alpha) \xrightarrow{d} N(0, \sigma^2_{\hat{\alpha}}) \tag{4.5}$$

with $\sigma^2_{\hat{\alpha}} \neq 0$. Then

$$\sqrt{T}(\hat{\alpha}^2 - \alpha^2) \xrightarrow{d} N(0, \sigma^2_{\hat{\alpha}^2}) \tag{4.6}$$

with $\sigma^2_{\hat{\alpha}^2} = 4\alpha^2\sigma^2_{\hat{\alpha}}$ which is obviously zero if $\alpha = 0$. Of course, this is a well-known result as in that case $T\hat{\alpha}^2$ is known to have a proper asymptotic

distribution and thus $\sqrt{T}\hat{\alpha}^2$ must be degenerate. Hence, estimated impulse responses may have a degenerate asymptotic distribution even if the underlying data generation process (DGP) is a well behaved stationary process.

One might be tempted to use (4.6) as a starting point for the construction of confidence intervals for α^2. Since the estimated $\sigma^2_{\hat{\alpha}^2}$ obtained by replacing α and $\sigma^2_{\hat{\alpha}}$ by their usual LS estimators will be nonzero almost surely one may consider the t-ratio $\sqrt{T}(\hat{\alpha}^2 - \alpha^2)/(2\hat{\alpha}\hat{\sigma}_{\hat{\alpha}})$ as a basis for constructing a CI. The next section shows that this results in a conservative CI for the case $\alpha = 0$. It is not clear that a conservative CI will always be obtained in the more interesting cases where impulse responses from higher dimensional processes are considered. Of particular concern is the fact that the procedure fails for a case of special interest, namely when the impulse responses are all zero. This failure is typical also for higher dimensional processes for which the order may also be greater than 1. Of course, the situation where some variable does not react to an impulse in some other variable, i.e. the impulse response is zero, is of particular interest because it means that there is no causal link in a certain part of the system. Hence, the asymptotic CIs fail in situations of particular importance. Note, however, that for stable, stationary VAR(p) processes, the asymptotic CIs work all right for $\phi_{ij,h}$ with $h \leq p$. This fact was used by Lütkepohl and Poskitt (1996) and Saikkonen and Lütkepohl (2000) to point out a possibility for circumventing the problem by assuming that the true DGP is an infinite order VAR process. Although the asymptotic problems can be fixed in this way, simulations reported in Lütkepohl and Poskitt (1996) indicate that this may not be very helpful in samples of the size typically available in macro econometrics.

4.1.2 Bootstrap inference

In the last two decades, bootstrap methods became popular in empirical work. The name *bootstrap* was introduced by Efron (1979). This work is widely seen as the starting point of the development and analysis of bootstrap methods in theoretical and empirical sciences which was accelerated by emerging computer power.

There exist different views about the main characterization of the bootstrap. One direction explains the bootstrap by replacing an unknown distribution function F by its empirical estimator $\widehat{F}$, in a functional form of an unknown quantity of interest. From this point of view, the estimation of the population mean by the sample mean is nothing else than applying the bootstrap Hall (1994). In some cases it is possible to calculate the bootstrap quantities by analytical means. But in the overwhelming number of cases (and particularly interesting cases) the analytical derivations are infeasible. Here, Monte–Carlo procedures are a convenient way to get an approximation for the bootstrap distribution.

Another view identifies bootstrap methods with procedures that apply Monte Carlo methods for numerical approximation (e.g. Davison and Hinkley (1997)). The main characterization of the bootstrap in this view is to rerun the experiment that led to the sample Y.

In both views the bootstrap is qualified as an asymptotic method. It does not provide the exact sample distribution of an estimator but its asymptotic distribution. This distribution is used as an approximation for the unknown sample distribution. Whether the approximation is good depends on the used sample size and on the functional form of the estimator.

Since derivations of the properties of bootstrap methods rely on asymptotic theory it should not come as a surprise that standard bootstrap techniques do not work well for some cases of interest here. In the next section implications

of these phenomena for constructing CIs based on asymptotic theory as well as the bootstrap are considered in detail. This is done for the simplest case of a stationary univariate AR(1) process. It is argued that methods that have problems in simple situations cannot be expected to work well for more general cases.

4.2 Confidence intervals for impulse responses from a univariate AR(1)

The analysis starts with standard methods which are followed by alternative methods. It is discussed whether modifications proposed in the literature help to circumvent the known problems. The entire section considers the process

$$y_t = \alpha y_{t-1} + u_t, \qquad t = 1, 2, \ldots$$

where the u_t's are i.i.d. with $E(u_t) = 0$ and $\sigma_{u_t}^2 = \sigma_u^2$. The impulse response coefficient at time horizon h is $\phi_h = \alpha^h$. The estimator for α is in the following denoted as $\widehat{\alpha}_T$, indicating that it is based on a sample size of T.

4.2.1 Confidence intervals based on first order asymptotic theory

Let $|\alpha| < 1$. It is well-known that

$$\sqrt{T}(\hat{\alpha}_T - \alpha) \xrightarrow{d} N(0, \sigma_{\hat{\alpha}_T}^2 = 1 - \alpha^2)$$

so that with $T^{-1}\sum_{t=1}^{T} y_{t-1}^2 \to \sigma_u^2/(1-\alpha^2)$ we have

$$\frac{\sqrt{\sum_{t=1}^{T} y_{t-1}^2}}{\sigma_u}(\widehat{\alpha}_T - \alpha) \xrightarrow{d} N(0,1),$$

see, for example, Anderson (1959). The standard approach uses $\widehat{\alpha}_T^h$ as a starting point for constructing a confidence interval for α^h. Let $\tilde{\sigma}_u^2 = T^{-1}\sum_{t=1}^{T}(y_t - \widehat{\alpha}_T y_{t-1})^2$. It is easy to see that $\tilde{\sigma}_u^2 = T^{-1}\sum u_t^2 + o_P(1) = \sigma_u^2 + o_P(1)$.

The asymptotic distribution of $\widehat{\alpha}_T^h - \alpha^h$ can be found by the so-called delta method (e.g. Goldberger (1991, p. 102), Lehmann (1999, pp. 85–93)) as indicated in (4.3) and (4.4). With $g(\alpha) = \alpha^h$

$$\sqrt{T}(g(\hat{\alpha}_T) - g(\alpha)) \xrightarrow{d} N\left(0, \left(\frac{\partial g(\alpha)}{\partial \alpha}\right)^2 \sigma^2_{\hat{\alpha}_T}\right) \tag{4.7}$$

$$= N\left(0, \left(h\alpha^{h-1}\right)^2 \sigma^2_{\hat{\alpha}_T}\right) \tag{4.8}$$

for $\alpha \neq 0$,

$$\frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widetilde{\sigma}_u h \widehat{\alpha}_T^{h-1}}(\widehat{\alpha}_T^h - \alpha^h) \xrightarrow{d} N(0,1). \tag{4.9}$$

This asymptotic result may be used to establish an asymptotic CI for α^h with a nominal coverage probability of $1-\gamma$. Let c_β denote the β-quantile of the standard normal distribution:

$$P\left(c_{(\gamma/2)} \leq \frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widetilde{\sigma}_u h \widehat{\alpha}_T^{h-1}}(\widehat{\alpha}_T^h - \alpha^h) \leq c_{(1-\gamma/2)}\right) = 1-\gamma$$

$$= P\left(\widehat{\alpha}_T^h - \frac{\widetilde{\sigma}_u h |\widehat{\alpha}_T^{h-1}|}{\sqrt{\sum_{t=1}^T y_{t-1}^2}} c_{(1-\gamma/2)} \leq \alpha^h \leq \widehat{\alpha}_T^h + \frac{\widetilde{\sigma}_u h |\widehat{\alpha}_T^{h-1}|}{\sqrt{\sum_{t=1}^T y_{t-1}^2}} c_{(1-\gamma/2)}\right).$$

The resulting confidence interval is

$$CI_1 = \left[\widehat{\alpha}_T^h - \frac{\widetilde{\sigma}_u h |\widehat{\alpha}_T|^{h-1}}{\sqrt{\sum_{t=1}^T y_{t-1}^2}} c_{(1-\gamma/2)}, \widehat{\alpha}_T^h + \frac{\widetilde{\sigma}_u h |\widehat{\alpha}_T|^{h-1}}{\sqrt{\sum_{t=1}^T y_{t-1}^2}} c_{(1-\gamma/2)}\right]. \tag{4.10}$$

It follows immediately from (4.9) that for $\alpha \neq 0$,

$$P\left(\alpha^h \in CI_1\right) \longrightarrow 1 - \gamma \quad \text{as} \quad T \to \infty, \tag{4.11}$$

that is, CI_1 has asymptotically the correct coverage probability.

However, for $\alpha = 0$, it turns out that

$$\frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widetilde{\sigma}_u h \widehat{\alpha}_T^{h-1}}(\widehat{\alpha}_T^h - \alpha^h) = \frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widetilde{\sigma}_u h}\widehat{\alpha}_T \xrightarrow{d} N(0, 1/h^2). \tag{4.12}$$

As a consequence CI_1 is conservative, with an asymptotic coverage probability larger than the prescribed $1-\gamma$. In terms of the length of the interval, CI_1

is about h times too large. Small sample errors in coverage probability are reported in the Monte-Carlo study in Section 4.3.

At this point some general comments are in order. The difficulty in getting asymptotically correct confidence intervals is caused by the fact that $\widehat{\alpha}_T^h - \alpha^h$ has a different limiting behavior for $\alpha \neq 0$ and $\alpha = 0$, respectively. In the first case $\sqrt{T}(\widehat{\alpha}_T^h - \alpha^h)$ has a nondegenerate limit distribution, whereas $T^{h/2}(\widehat{\alpha}_T^h - \alpha^h)$ has a proper limit distribution in the latter case. This change in the limiting behavior is not fully captured by the factor $\sqrt{\sum_{t=1}^T y_{t-1}^2}/(\widetilde{\sigma}_u h \widehat{\alpha}_T^{h-1})$ that leads to a pivotal statistic only in the case $\alpha \neq 0$.

Such a situation is already known for $\widehat{\alpha}_T - \alpha$ for the critical case $|\alpha| = 1$. The three cases, $|\alpha| < 1$, $|\alpha| = 1$ and $|\alpha| > 1$, lead to very different limit distributions.

Hence, if the parameter space is extended and also allows for nonstationary processes the problem of incorrect CIs arises also in other situations than the simple one considered in detail in the foregoing. Of course, the problem also becomes more severe when higher order and higher dimensional processes are considered. In the following the focus is exclusively on stationary univariate AR(1) processes. However, it should be clear that similar problems also arise in other situations.

4.2.2 Confidence intervals based on the standard bootstrap

This chapter considers the same bootstrap method as implemented in JMulTi. In the notation of the simplified case analyzed here it reads:

1) Estimate $\widehat{\alpha}_T$ by least squares.

2) Generate bootstrap residuals $u_1^*, \ldots, u_T^*$ by randomly drawing with replacement from the set of estimated and re-centered residuals, $\{\widehat{u}_1 -$

$\bar{u}_\cdot, \ldots, \widehat{u}_T - \bar{u}_\cdot\}$, where $\widehat{u}_t = y_t - \widehat{\alpha}_T y_{t-1}$, and $\bar{u}_\cdot = T^{-1} \sum \widehat{u}_t$.

3) Set $y_0^* = y_0$ and construct bootstrap time series recursively by

$$y_t^* = \widehat{\alpha}_T y_{t-1}^* + u_t^*, \quad t = 1, \ldots, T. \tag{4.13}$$

4) Calculate a bootstrap version of the statistic of interest, in this case

$$\widehat{\alpha}_T^* = \sum_{t=1}^{T} y_t^* y_{t-1}^* / \sum_{t=1}^{T} (y_{t-1}^*)^2.$$

A slightly different method was proposed by Efron and Tibshirani (1986) who centered the original data $\{y_t\}$ first, rather than centering the estimated residuals. Such a scheme was also proposed by de Wet and van Wyk (1986) in the context of a linear regression model, where the errors were assumed to be generated by a linear AR(1) process.

The technique of computing bootstrap confidence intervals established by Efron and Tibshirani (1993) is presented. The bootstrap impulse responses $(\widehat{\alpha}_T^*)^h$ are derived in the same manner as described above. Now, let $s_{\gamma/2}^*$ and $s_{(1-\gamma/2)}^*$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of $\mathcal{L}((\widehat{\alpha}_T^*)^h \mid y_0, \ldots, y_T)$, respectively, and define

$$\widetilde{CI}_S = \left[s_{\gamma/2}^*, s_{(1-\gamma/2)}^* \right].$$

The interval $\widetilde{CI}_S$ is the percentile confidence interval described by Efron and Tibshirani (1993). However, in the latter reference it is pointed out that $\widetilde{CI}_S$ may not have the desired coverage. This is for example the case if $\widehat{\alpha}_T^h$ is a biased estimator of α^h. To fix this drawback different modifications of $\widetilde{CI}_S$ were proposed in the literature. One possibility is to use a bias correction based on $(E\widehat{\alpha}_T - \alpha)$ and another possibility is to consider $(E\widehat{\alpha}_T^h - \alpha^h)$. The latter type of bias correction was introduced in Efron and Tibshirani (1993). In this study the methodology of Kilian (1998b) is employed that takes into

account the bias terms of both the initial estimation ($E\widehat{\alpha}_T - \alpha$) and of the bootstrap estimation ($E\widehat{\alpha}^*_T - \widehat{\alpha}_T$). In Kilian (1998b) it is argued that each time α is estimated by least squares the estimator $\widehat{\alpha}_T$ is biased. It is therefore proposed to bias correct the least squares estimator each time it is computed.

The resulting bias corrected estimator will be denoted by $\widetilde{\alpha}_T$. Kilian (1998b) proposes to estimate the bias of $\widehat{\alpha}_T$ by $\widehat{bias} = T^{-1}\sum_{i=1}^{T}\widehat{\alpha}^{*,i}_T - \widehat{\alpha}_T$, which is then used to derive the bias corrected estimator $\widetilde{\alpha}_T$. If $|\widehat{\alpha}_T| \geq 1$, set $\widetilde{\alpha}_T = \widehat{\alpha}_T$ without any adjustments. If $|\widehat{\alpha}_T| < 1$ the bias corrected estimate is $\widetilde{\alpha}_T = \widehat{\alpha}_T - \widehat{bias}$. If $|\widetilde{\alpha}_T| \geq 1$, set $\widetilde{\alpha}_T$ to a value smaller than but close to one. In the Monte-Carlo experiment in Section 4.3 we use $\widetilde{\alpha}_T = .99$ for the latter case. The intention of this correction is to circumvent pushing stationary estimates into the nonstationary region. Then the aforementioned resampling plan is applied with $y^*_t = \widetilde{\alpha}_T y^*_{t-1} + u^*_t, \quad t = 1, \ldots, T$, instead of formula (4.13). The resulting $\widehat{\alpha}^*_T$ are bias corrected as $\widetilde{\alpha}^*_T = \widehat{\alpha}^*_T - \widehat{bias}^*$ with $\widehat{bias}^* = T^{-1}\sum_{i=1}^{T}\widehat{\alpha}^{**,i}_T - \widehat{\alpha}^*_T$. In practice $\widehat{bias}^*$ is not computed but $\widehat{bias}$ is used as an approximation.

Now, let $t^*_{\gamma/2}$ and $t^*_{(1-\gamma/2)}$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of $\mathcal{L}((\widetilde{\alpha}^*_T)^h \mid y_0, \ldots, y_T)$, respectively, and define

$$CI_S = \left[t^*_{\gamma/2}, t^*_{(1-\gamma/2)}\right].$$

Unfortunately, even with bias correction this confidence interval may be problematic for the purposes considered here. The reason is that an estimator $\widetilde{\alpha}^*_T$ which is identically equal to zero is very unlikely. Hence, for even h, $(\widetilde{\alpha}^*_T)^h$ is almost always strictly greater than zero. Hence the bootstrap distribution in this case has positive support and a confidence interval based on the quantiles of this distribution will not include zero. Consequently, for $\alpha = 0$ and thus $\alpha^h = 0$, the coverage probability of such an interval will be zero in small samples and asymptotically. This phenomenon will become apparent in the simulations in Section 4.3.

4.2.3 Hall's percentile method

Another method of deriving confidence intervals was described by Hall (1992). Suppose the resampling plan presented at the beginning of the previous subsection is used and let $t^*_{\gamma/2}$ and $t^*_{(1-\gamma/2)}$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of $\mathcal{L}((\widehat{\alpha}^*_T)^h - \widehat{\alpha}^h_T \mid y_0, \ldots, y_T)$, respectively. According to the usual bootstrap analogy, $\mathcal{L}(\widehat{\alpha}^h_T - \alpha^h) \approx \mathcal{L}((\widehat{\alpha}^*_T)^h - \widehat{\alpha}^h_T \mid y_0, \ldots, y_T)$, one gets the interval

$$CI_H \;=\; \left[\widehat{\alpha}^h_T \;-\; t^*_{(1-\gamma/2)}, \widehat{\alpha}^h_T \;-\; t^*_{\gamma/2}\right].$$

Hall (1992) calls such intervals "percentile intervals". Therefore, in the following the method leading to CI_H is referred to as Hall's percentile method, whereas the method of the previous subsection is called Efron's percentile method.

Benkwitz et al. (2000) show that for $\alpha \neq 0$ and $|\alpha| < 1$,

$$P_\alpha\left(\alpha^h \in CI_H\right) \longrightarrow 1 \;-\; \gamma \tag{4.14}$$

but for the interesting case $\alpha = 0$ the statement (4.14) does not hold. Moreover, it is argued that usual small sample corrections which aim at reducing the bias do not help in this context.

It is well-known that a general statistic μ_T can be better approximated by the bootstrap if it depends to a lesser extent on the unknown distribution that governs the data generating process. For example, concerning the sample mean of i.i.d. random variables it is well known that studentizing leads to a better rate of approximation by the bootstrap; see Hall (1988). Therefore, the statistic $(\widehat{\alpha}^h_T - \alpha^h)/\sqrt{\widehat{\mathrm{var}}(\widehat{\alpha}^h_T)}$ is used as a basis for the construction of a confidence interval, and determine a bootstrap quantile based on the statistic $((\widehat{\alpha}^*_T)^h - \widehat{\alpha}^h_T)/\sqrt{\widehat{\mathrm{var}}((\widehat{\alpha}^*_T)^h)}$. The variances were estimated by the bootstrap method, that is,

$$\widehat{\mathrm{var}}(\widehat{\alpha}^h_T) \;=\; \frac{1}{B^*}\sum_{i=1}^{B^*}(\widehat{\alpha}^{*,i}_T)^{2h} \;-\; \left[\frac{1}{B^*}\sum_{i=1}^{B^*}(\widehat{\alpha}^{*,i}_T)^{h}\right]^2$$

and

$$\widehat{\mathrm{var}}((\widehat{\alpha}_T^*)^h) \;=\; \frac{1}{B^{**}}\sum_{i=1}^{B^{**}}(\widehat{\alpha}_T^{**,i})^{2h} \;-\; \left[\frac{1}{B^{**}}\sum_{i=1}^{B^{**}}(\widehat{\alpha}_T^{**,i})^{h}\right]^2,$$

where B^* and B^{**} are the respective numbers of bootstrap replications. Note in particular that $\widehat{\alpha}_T^{**,i}$ is obtained by a double bootstrap, that is, pseudo-data are generated according to a process with the parameter $\widehat{\alpha}_T^*$.

Let $t^{**}_{\gamma/2}$ and $t^{**}_{(1-\gamma/2)}$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles, respectively, of $\mathcal{L}\left([(\widehat{\alpha}_T^*)^h - \widehat{\alpha}_T^h]/\sqrt{\widehat{\mathrm{var}}((\widehat{\alpha}_T^*)^h)} \mid y_0,\ldots,y_T\right)$. Based on the studentized statistics, the interval

$$CI_{SH} \;=\; \left[\widehat{\alpha}_T^h - t^{**}_{(1-\gamma/2)}\sqrt{\widehat{\mathrm{var}}(\widehat{\alpha}_T^h)},\; \widehat{\alpha}_T^h - t^{**}_{\gamma/2}\sqrt{\widehat{\mathrm{var}}(\widehat{\alpha}_T^h)}\right]$$

is obtained. However, although studentizing improves the accuracy of the bootstrap in many "regular"cases, this is not the case here if $\alpha = 0$ because, for $\alpha = 0$ and $h > 1$, the distributions of $(\widehat{\alpha}_T^h - \alpha^h)$ and $((\widehat{\alpha}_T^*)^h - \widehat{\alpha}_T^h)$ are of different type, and hence, $(\widehat{\alpha}_T^h - \alpha^h)/\sqrt{\widehat{\mathrm{var}}(\widehat{\alpha}_T^h)}$ and $((\widehat{\alpha}_T^*)^h - \widehat{\alpha}_T^h)/\sqrt{\widehat{\mathrm{var}}((\widehat{\alpha}_T^*)^h)}$ do not coincide asymptotically. In the simulations reported later a closer look will be taken at the performance of the latter bootstrap CI.

4.2.4 Confidence intervals based on a superefficient estimator

The main reason why the standard bootstrap fails at the point $\alpha = 0$ is that $P_{\alpha_T}\left(T^{h/2}(\widehat{\alpha}_T^h - \alpha_T^h) \le x\right)$ remains different from $P_0\left(T^{h/2}(\widehat{\alpha}_T^h - \alpha^h) \le x\right)$, even if the sequence $\{\alpha_T\}$ tends to 0 with the rate $T^{-1/2}$. Since $\widehat{\alpha}_T$ converges to the true value just with this rate, the bootstrap is not able to recognize the presence of the case $\alpha = 0$. A well-known remedy to such problems with singularities in the limit distribution is the use of a so-called superefficient estimator that converges at a faster rate just at these critical points in the parameter space. Datta and Sriram (1997) used this idea to devise a bootstrap for AR(1) processes that estimates $\mathcal{L}(\widehat{\alpha}_T - \alpha)$ consistently for all $\alpha \in \mathbf{R}$.

Whereas Datta and Sriram (1997) used an estimator that is superefficient at $\alpha = \pm 1$, this property is needed here for $\alpha = 0$. Let $\{c_T\}$ be any sequence satisfying $c_T \to 0$ and $T^{1/2}c_T \to \infty$ as $T \to \infty$. Then the threshold estimator

$$\widetilde{\alpha}_T = \begin{cases} \widehat{\alpha}_T, & \text{if } |\widehat{\alpha}_T| > c_T \\ 0 & \text{otherwise} \end{cases} \tag{4.15}$$

is superefficient at $\alpha = 0$, that is, $\widetilde{\alpha}_T$ converges with a faster rate than $T^{-1/2}$ to the true value. This estimator allows to switch between the two cases, $\alpha = 0$ and $\alpha \neq 0$. Let:

$$\begin{aligned} S_T &= \frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widetilde{\sigma}_u[\widehat{\alpha}_T^{h-1} + (h-1)\widetilde{\alpha}_T^{h-1}]}(\widehat{\alpha}_T^h - \alpha^h) \\ &= \begin{cases} \frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widetilde{\sigma}_u[h\widehat{\alpha}_T^{h-1} + o_P(1)]}(\widehat{\alpha}_T^h - \alpha^h), & \text{if } \alpha \neq 0 \\ \frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widetilde{\sigma}_u[\widehat{\alpha}_T^{h-1} + o_P(T^{-1/2})]}(\widehat{\alpha}_T^h - \alpha^h), & \text{if } \alpha = 0 \end{cases} . \end{aligned} \tag{4.16}$$

Obviously, $S_T \xrightarrow{d} N(0,1)$ for all $|\alpha| < 1$. Therefore,

$$\begin{aligned} CI_5 = \Bigg[&\widehat{\alpha}_T^h - \frac{\widetilde{\sigma}_u|\widehat{\alpha}_T^{h-1} + (h-1)\widetilde{\alpha}_T^{h-1}|}{\sqrt{\sum_{t=1}^T y_{t-1}^2}} c_{(1-\gamma/2)}, \\ &\widehat{\alpha}_T^h - \frac{\widetilde{\sigma}_u|\widehat{\alpha}_T^{h-1} + (h-1)\widetilde{\alpha}_T^{h-1}|}{\sqrt{\sum_{t=1}^T y_{t-1}^2}} c_{\gamma/2} \Bigg] \end{aligned} \tag{4.17}$$

is a confidence interval for α^h with an asymptotic level $1 - \gamma$, that is,

$$P_\alpha\left(\alpha^h \in CI_5\right) \longrightarrow 1 - \gamma \quad \text{for all } |\alpha| < 1. \tag{4.18}$$

A closer look at the proposed procedure indicates that in this case the convergence is not uniform in α. Benkwitz et al. (2000) point out that (4.18) fails, if instead of any *fixed* α a sequence $\{\alpha_T\}$ tending to zero at a slightly slower rate than $T^{-1/2}$ is considered.

Rather than relying on the asymptotic distribution, a bootstrap approximation of $\mathcal{L}(\widehat{\alpha}_T^h - \alpha^h)$ in connection with the above superefficient estimator

$\widetilde{\alpha}_T$ could be used. This was done by Datta and Sriram (1997) for estimating the distribution of $\widehat{\alpha}_T - \alpha$ around $|\alpha| = 1$. For this approach no uniform convergence results are available either. Of course, for practical purposes one may be satisfied with pointwise convergence. Even then it will be difficult to generalize this approach to higher order and higher dimensional processes because it requires that care has to be taken for every possible singularity point of the asymptotic distribution. In general this may be a difficult or impossible task. The use of superefficient estimators can solve problems with different limit distributions at *known isolated* points in the parameter space. Since any estimator can only be superefficient on sets with measure 0, it is impossible to apply such a strategy in the case of rapidly changing limit distributions, where these changes occur at *unknown* points in the parameter space. Therefore other procedures have been considered which do not require the user to identify all possible singularity points prior to using the bootstrap. One such procedure will be described in the following subsection.

4.2.5 Subsampling

Subsampling is a relatively new technique that aims at improving the relation between the rate of convergence of the bootstrap version of the estimator and the rate of convergence of the parameter that controls the DGP in the bootstrap world. It is characterized by resampling fewer observations than contained in the observed sample ($N < T$). Surveys on this technique are given by Bertail et al. (1999), in the discussion to Li and Maddala (1996), and by Bickel et al. (1997).

Subsampling is straightforward, if the rate of convergence is constant over the whole parameter space and if only the shape of the corresponding limit distributions is different. Although recent work of Bertail et al. (1999) also allows for the case of different rates of convergence to be estimated separately,

these complications are avoided here by multiplying the statistic of interest, $\widehat{\alpha}_T^h - \alpha^h$, with an appropriate normalizing factor. This was also done by Datta (1996) and Heimann and Kreiss (1996) in the case of estimating the distribution of $(\mathrm{var}(\widehat{\alpha}_T))^{-1/2}(\widehat{\alpha}_T - \alpha)$ around $|\alpha| = 1$. The normalizing factor for all $|\alpha| < 1$ turns out to be

$$\frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widehat{\alpha}_T^{h-1}}$$

(see Benkwitz et al. (2000)).

Let $t^*_{\gamma/2}$ and $t^*_{(1-\gamma/2)}$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of the distribution of

$$\frac{\sqrt{\sum_{t=1}^N (y_{t-1}^*)^2}}{(\widehat{\alpha}_N^*)^{h-1}} \left((\widehat{\alpha}_N^*)^h - \widehat{\alpha}_T^h \right),$$

respectively. Then

$$P\left(t^*_{\gamma/2} < \frac{\sqrt{\sum_{t=1}^T y_{t-1}^2}}{\widehat{\alpha}_T^{h-1}} (\widehat{\alpha}_T^h - \alpha^h) < t^*_{(1-\gamma/2)} \right) \longrightarrow 1-\gamma,$$

which implies that

$$CI_6 = \left[\widehat{\alpha}_T^h - t^*_{(1-\gamma/2)} \frac{|\widehat{\alpha}_T|^{h-1}}{\sqrt{\sum_{t=1}^T y_{t-1}^2}}, \;\; \widehat{\alpha}_T^h - t^*_{\gamma/2} \frac{|\widehat{\alpha}_T|^{h-1}}{\sqrt{\sum_{t=1}^T y_{t-1}^2}} \right]$$

is an asymptotic $(1-\gamma)$-confidence interval for $|\alpha| < 1$.

In more general situations where higher order and higher dimensional processes are considered it will not be easy to find a suitable normalization of estimated impulse responses analogous to the factor $\sqrt{\sum_{t=1}^T y_{t-1}^2}/\widehat{\alpha}_T^{h-1}$ in this simple case which guarantees a constant rate of convergence. In the next subsection a subsampling approach is presented which is theoretically suitable in such a case. It may be computationally quite demanding, however.

4.2.6 Subsampling with estimated rate of convergence

The problem with the subsampling procedure of the previous subsection is that it may be difficult to find a quantity with constant rate of convergence in all of the feasible parameter space. For this situation, Bertail et al. (1999) proposed to estimate the rate of convergence, τ_T, say.

The convergence rate is estimated using two subsampling distributions based on the subsample sizes $N_1 = N_1(T)$ and $N_2 = N_2(T)$. If $N_1, N_2 \to \infty$ as $T \to \infty$

$$\left.\begin{array}{l} P(\tau_{N_1}((\widehat{\alpha}^*_{N_1})^h - \widehat{\alpha}^h_T) \le x|\underline{Y}) \\ P(\tau_{N_2}((\widehat{\alpha}^*_{N_2})^h - \widehat{\alpha}^h_T) \le x|\underline{Y}) \end{array}\right\} = F_\infty(x) + o_P(1), \tag{4.19}$$

where $\underline{Y}$ denotes the sample $(y_0, \ldots, y_T)$ and F_∞ the limit distribution of $\tau_{N_i}((\widehat{\alpha}_{N_i})^h - \widehat{\alpha}^h_T)$. The quantile functions, that is, the inverses of the cumulative distribution functions fulfill

$$F^{-1}_{\tau_{N_i}((\widehat{\alpha}^*_{N_i})^h - \widehat{\alpha}^h_T)}(x|\underline{Y}) = \tau_{N_i} F^{-1}_{((\widehat{\alpha}^*_{N_i})^h - \widehat{\alpha}^h_T)}(x|\underline{Y}), \quad i = 1, 2. \tag{4.20}$$

Since F_∞ is continuous, one can derive from equations (4.19) and (4.20)

$$F^{-1}_{((\widehat{\alpha}^*_{N_i})^h - \widehat{\alpha}^h_T)}(x|\underline{Y}) = \tau^{-1}_{N_i} F^{-1}_\infty(x) + o_P(\tau^{-1}_{N_i}), \quad i = 1, 2. \tag{4.21}$$

Let x_{2j}, for $j = 1, \ldots, J$, be some points in the interval $(0.5, 1)$ and x_{2j-1}, for $j = 1, \ldots, J$, be some points in the interval $(0, 0.5)$. Let

$$a^{N_i}_j = \log[\, F^{-1}_{((\widehat{\alpha}^*_{N_i})^h - \widehat{\alpha}^h_T)}(x_{2j}|\underline{Y}) - F^{-1}_{((\widehat{\alpha}^*_{N_i})^h - \widehat{\alpha}^h_T)}(x_{2j-1}|\underline{Y})\,],$$

for $i = 1, 2$, and $a^\infty_j = \log[\, F^{-1}_\infty(x_{2j}) - F^{-1}_\infty(x_{2j-1})\,]$ be the logarithm of the j-th interquantile range of the subsampling distribution and the limit distribution, respectively. From (4.21) it follows that

$$a^{N_i}_j = \log(\tau^{-1}_{N_i}) + a^\infty_j + o_P(1), \quad j = 1, \ldots, J.$$

Let δ be such that $\tau_T = T^\delta$ and thus $\tau_{N_i} = N_i^\delta$. Hence, assuming $0 < N_2 < N_1 < T$ one can obtain $\delta = (\, a^{N_2}_j - a^{N_1}_j\,) / (\, \log(N_1) - \log(N_2)\,) + o_P(1)$. Therefore, $\widehat{\delta}_j = (\, a^{N_2}_j - a^{N_1}_j\,) / (\, \log(N_1) - \log(N_2)\,)$ is defined and δ is estimated by

$\widehat{\delta} = J^{-1} \sum_{j=1}^{J} \widehat{\delta}_j$. It follows from the consistency established in (4.19) that $\widehat{\delta}$ is a consistent estimator of δ. The estimator of τ_T is obtained as $\widehat{\tau}_T = T^{\widehat{\delta}}$.

Now it is possible to proceed with constructing confidence intervals using subsampling as introduced in Section 4.2.5. The difference in both methods is found in the norming factor for the statistic $((\widehat{\alpha}_N^*)^h - \widehat{\alpha}_T^h)$. Here, the norming factor is estimated for each α^h separately, whereas the method in Section 4.2.5 uses algebraic manipulation to handle the problem of different convergence rates. The latter approach implies that for every statistic new analytical work is required for this manipulation.

In the present approach where the convergence rate is estimated, the unknown distribution of interest, $\mathcal{L}(\widehat{\tau}_T(\widehat{\alpha}_T^h - \alpha^h))$, is approximated by

$$\mathcal{L}\left(\widehat{\tau}_{N(T)}((\widehat{\alpha}_{N(T)}^*)^h - \widehat{\alpha}_T^h) | \underline{Y}\right), \tag{4.22}$$

with subsample size $N(T)$. Let $t_{\gamma/2}^*$ and $t_{1-\gamma/2}^*$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of (4.22), respectively. Then

$$P\left(t_{\gamma/2}^* < \widehat{\tau}_T(\widehat{\alpha}_T^h - \alpha^h) < t_{1-\gamma/2}^*\right) \to (1-\gamma).$$

Hence,

$$CI_7 = \left[\widehat{\alpha}_T^h - \frac{t_{1-\gamma/2}^*}{\widehat{\tau}_T}, \;\; \widehat{\alpha}_T^h - \frac{t_{\gamma/2}^*}{\widehat{\tau}_T}\right]$$

is a confidence interval which has asymptotically the correct coverage probability of $(1-\gamma)$.

4.2.7 Indirect confidence intervals

The next method of constructing confidence intervals uses ideas of Sims and Zha (1999) and may be motivated as follows. Assume for a moment that the distribution of the innovations u_t is exactly known. Then, for each fixed α, the

(hypothetical) distribution of $\widehat{\alpha}_T^h$ can be calculated. Let $t_{\alpha,\gamma/2}$ and $t_{\alpha,(1-\gamma/2)}$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of the corresponding distribution. Define

$$\widetilde{CI}_8 \;=\; \left\{\alpha^h \mid \;\; \widehat{\alpha}_T^h \in [t_{\alpha,\gamma/2}, t_{\alpha,(1-\gamma/2)}]\right\}.$$

By construction,

$$P_\alpha\left(\alpha^h \in \widetilde{CI}_8\right) \;=\; P_\alpha\left(\widehat{\alpha}_T^h \in [t_{\alpha,\gamma/2}, t_{\alpha,(1-\gamma/2)}]\right) \;=\; 1 \;-\; \gamma,$$

that is, $\widetilde{CI}_8$ is an *exact* confidence set for α^h, for all values of $\alpha \in \mathbf{R}$. This approach was proposed by Andrews (1993, Section 4) in the case of a known distribution of the innovations.

Since the distribution of the innovations u_t is usually unknown, we propose to estimate it by the bootstrap. Let $u_1^*, \ldots, u_T^*$ be drawn with replacement from $\{\widehat{u}_1 - \bar{u}_\cdot, \ldots, \widehat{u}_T - \bar{u}_\cdot\}$, where $\widehat{u}_t = y_t - \widehat{\alpha}_T y_{t-1}$, as before. For each value of α a (hypothetical) bootstrap process is generated. It is based on the model equation

$$y_t^{\alpha,*} \;=\; \alpha y_{t-1}^{\alpha,*} \;+\; u_t^*, \quad t = 1, \ldots, T.$$

Now the behavior of $\widehat{\alpha}_T$ under each hypothetical value of α can be imitated by

$$\widehat{\alpha}_T^{\alpha,*} = \sum_{t=1}^{T} y_t^{\alpha,*} y_{t-1}^{\alpha,*} / \sum_{t=1}^{T} (y_{t-1}^{\alpha,*})^2.$$

Let $t^*_{\alpha,\gamma/2}$ and $t^*_{\alpha,(1-\gamma/2)}$ be the $\gamma/2$- and $(1-\gamma/2)$-quantiles of $\mathcal{L}((\widehat{\alpha}_T^{\alpha,*})^h)$, respectively. According to the theoretical set $\widetilde{CI}_8$ above, an asymptotic confidence set is constructed as

$$CI_8 \;=\; \left\{\alpha^h \mid \;\; \widehat{\alpha}_T^h \in [t^*_{\alpha,\gamma/2}, t^*_{\alpha,(1-\gamma)/2}]\right\}. \tag{4.23}$$

The confidence set CI_8 is not necessarily an interval. However, Benkwitz et al. (2000) proofed that CI_8 will consist of one large connected set plus perhaps some additional sets of asymptotically negligible size. Since this "indifference region" is asymptotically negligible, instead of CI_8 the smallest interval which

contains the complete confidence set, say $\overline{CI}_9$ is used. Is is also shown that asymptotically both intervals have the correct probability content and the convergence is in fact quite rapid. Again this method becomes rather complicated and computer intensive in more general situations with higher order and higher dimensional processes. However, in principle it can be extended.

4.3 A Monte-Carlo experiment

The Monte-Carlo experiment illustrates the absolute and relative performance of the statistical methods explained in the previous section. Monte-Carlo studies are in general suited to provide more insights into the properties of statistical procedure. They can also provide counter examples. However, such studies cannot replace proofs since they operate on isolated points in the parameter space only.

4.3.1 Design

The Monte-Carlo experiment compares the performance of CI_S, CI_H, CI_{SH}, and CI_1, CI_5 – CI_8 for estimated impulse responses at response horizons 1, 2, 3, and 4. The reason why higher response horizons are omitted from this experiment is the missing numerical accuracy of computer software in extreme cases. Nevertheless, the performance is perfectly illustrated also for the chosen 'low' forecast horizon.

The performance is evaluated by estimating the real coverage probability, p, of nominal 95% confidence intervals and by looking at the length of the confidence intervals, l.

Data was artificially generated by the univariate AR(1) process

$$y_t = \alpha y_{t-1} + u_t, \qquad t = 1, \ldots, 100, \quad y_0 = 0, \quad u_t \sim N(0,1), \quad \alpha = 0, 0.2, 0.5, 0.9, 0.99 \tag{4.24}$$

Table 4.1: Average threshold c_T of the superefficient estimator for sample size $T = 100$. Numbers in parentheses are standard deviations.

$\alpha = 0$	$\alpha = .20$	$\alpha = .50$	$\alpha = .90$	$\alpha = .99$
.305	.299	.266	.143	.074
(.021)	(.022)	(.024)	(.001)	(.001)

For each α, $M = 1,000$ Monte Carlo (MC) replications were performed. If $E(\widehat{p}) = p = 0.95$, the standard error of $\widehat{p}$ is $\sqrt{p(1-p)/M} \approx 0.007$.

The bootstrap distributions were approximated using 2,000 bootstrap drawings. The threshold c_T for the superefficient estimator in Section 4.1 is chosen to be

$$c_T = \frac{\sqrt{2 \log T}}{\sqrt{\sum_{t=1}^{T} y_t^2}}.$$

While the denominator is just a scaling factor the particular choice of the numerator, $\sqrt{2 \log T}$, is motivated by the fact that a standard normal random variable exceeds this bound in absolute value with a probability of about T^{-1}, which is considered as a sufficient value. The actual thresholds were filed and are reported in Table 4.1.

For CI_{SH} the variance of $(\widehat{\alpha}_T^*)^h$ is estimated with $B^{**} = 50$ bootstrap replications. The subsample length for CI_6 is $N(T) = 90$ for $T = 100$. Finding a suitable subsample size is actually a difficult task. Politis and Romano (1994) found the order $N(T) \asymp T^{2/3}$ to be optimal on the basis of second-order asymptotic theory. Different values for $N(T)$ including the proposed ones were tried in the experiment. The results showed that the estimated coverage probability does not change much when changing $N(T)$. However, for $T = 100$ it is found that "small" $N(T)$ (e.g. 25 or 30) resulted in confidence intervals of greater length than rather "large" values for $N(T)$ (e.g. 80 or 90). Whereas

Table 4.2: Average coefficient of the rate of convergence for the distribution of $(\widehat{\alpha}_T^*)^h - \widehat{\alpha}_T^h$. Empirical standard deviation in parentheses. Note that the theoretical value is 1/2 for $\alpha \neq 0$ (independent of h), and $h/2$ for $\alpha = 0$. Sample size $T = 100$.

	$\alpha = 0$	$\alpha = .20$	$\alpha = .50$	$\alpha = .90$	$\alpha = .99$
$h = 1$	.486	.487	.492	.565	.745
	(.041)	(.041)	(.043)	(.062)	(.116)
$h = 2$	.742	.519	.455	.535	.719
	(.197)	(.161)	(.041)	(.060)	(.115)
$h = 3$	1.025	0.671	.451	.508	.696
	(.312)	(.255)	(.050)	(.058)	(.114)
$h = 4$	1.463	.902	.468	.486	.675
	(.425)	(.403)	(.071)	(.056)	(.114)

this phenomenon was not really observable for $\alpha = 0$, the enlargement of the confidence intervals became quite substantial for α greater than or equal to 0.5. Therefore, $N(T) = 90$ was chosen in the current setting.

For the estimation of the rate of convergence, CI_7 uses two subsample sizes $N_1 = 80$ and $N_2 = 30$, and the interquantile ranges of $J = 4$ pairs $(x_{2j-1}, x_{2j}) = \{(.10, .90), (.15, .85), (.20, .80), (.25, .75)\}$. The results are reported in Table 4.2 where it is seen that, although the theoretical convergence rate for all $\alpha \neq 0$ is the same, the estimated coefficient associated with the rate of convergence for $\alpha = 0.2$ is an intermediate value between the rate for $\alpha = 0$ and $\alpha = .5$ whereas the coefficient for $\alpha = .5$ and .9 is similar to the theoretical coefficient of $\delta = .5$. For $\alpha = .99$ a slightly larger value of δ is usually estimated which reflects the true coefficient of $\delta = 1$ for nonstationary processes with $\alpha = 1$. The confidence intervals CI_7 are constructed with subsample sizes $N(T)$=50.

Again several subsample sizes were tried and it was found that the chosen value performs best in the current setting.

A precise coverage probability is considered as the most important feature of a confidence interval. However, from the point of view of the usefulness of the intervals, their average length is also an important factor. These two points are discussed in the following. Possible generalizations of the methods to multivariate and higher order autoregressive processes are discussed afterwards. The previous section showed that CI_5 to CI_8 are asymptotically correct for all α, whereas CI_1, CI_S, CI_H, and CI_{SH} are only correct if $\alpha \neq 0$. Problems may also arise for α values close to the nonstationary region, that is, for α close to one.

4.3.2 Results

In Tables 4.3 to 4.6 the estimated real coverage and estimated interval length are reported. Efron's percentile (BC) stands for Efron's percentile method (bias corrected). The bias correction was carried out using the method of Kilian (1998b).

The following observations emerge from Tables 4.3 – 4.6. First, all methods lead to nearly identical results in the case of $h = 1$, even for $\alpha = 0$. For larger h, the indirect method is overall the most precise in terms of coverage. As expected it produces the nominal coverage level almost exactly for all cases considered. The average length of these CIs is in some cases a bit larger than that of other methods though. The main disadvantage of the indirect method is the difficulty to extend it to higher dimensional and higher order cases.

For the critical case where $\alpha = 0$, the CIs based on first–order asymptotics are clearly conservative for $h > 1$ and have a considerably larger coverage probability than the nominal 95%. The length of the intervals is surprisingly small given that it was found in the asymptotic analysis that even further

Table 4.3: Impulse response of order $h = 1$ ($\phi_1 = \alpha$), estimated coverage probability ($\widehat{p}$) and average length ($\widehat{l}$) of estimated 95% CIs, sample size $T = 100$.

		$\alpha = 0$		$\alpha = .20$		$\alpha = .50$		$\alpha = .90$		$\alpha = .99$	
		$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$
First-order asymptotic	(CI_1)	.945	.392	.947	.384	.948	.342	.951	.184	.936	.096
Efron's percentile (BC)	(CI_S)	.948	.387	.945	.380	.950	.340	.952	.190	.941	.118
Hall's percentile	(CI_H)	.934	.388	.930	.380	.932	.341	.889	.203	.931	.141
Studentized	(CI_{SH})	.942	.399	.942	.392	.945	.348	.915	.185	.928	.109
Superefficient est.	(CI_5)	.945	.392	.947	.384	.948	.342	.951	.184	.936	.096
Subsampling	(CI_6)	.939	.381	.935	.374	.939	.332	.934	.177	.928	.091
Subsampling with $\widehat{\tau}_T$	(CI_7)	.931	.381	.926	.375	.916	.337	.878	.202	.951	.141
Indirect	(CI_8)	.956	.399	.956	.391	.949	.348	.949	.191	.953	.107

Table 4.4: Impulse response of order $h = 2$ ($\phi_2 = \alpha^2$), estimated coverage probability ($\widehat{p}$) and average length ($\widehat{l}$) of estimated 95% CIs, sample size $T = 100$.

		$\alpha = 0$		$\alpha = .20$		$\alpha = .50$		$\alpha = .90$		$\alpha = .99$	
		$\widehat{p}$	$\widehat{l}$ $\times 10^{-2}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$
First-order asymptotic	(CI_1)	1.000	6.257	.874	.152	.932	.332	.937	.322	.927	.184
Efron's percentile (BC)	(CI_S)	.000	7.951	.945	.152	.950	.327	.952	.328	.941	.222
Hall's percentile	(CI_H)	.983	7.799	.701	.147	.873	.316	.881	.340	.926	.259
Studentized	(CI_{SH})	.979	5.298	.788	.172	.956	.376	.920	.336	.933	.217
Superefficient est.	(CI_5)	.946	3.136	.754	.096	.932	.330	.937	.322	.927	.184
Subsampling	(CI_6)	1.000	0.121	.905	.262	.977	.356	.935	.321	.934	.181
Subsampling with $\widehat{\tau}_T$	(CI_7)	1.000	6.187	.682	.119	.862	.292	.866	.332	.937	.255
Indirect	(CI_8)	.951	7.816	.954	.163	.953	.344	.949	.337	.953	.208

Table 4.5: Impulse response of order $h = 3$ ($\phi_3 = \alpha^3$), estimated coverage probability ($\widehat{p}$) and average length ($\widehat{l}$) of estimated 95% CIs, sample size $T = 100$.

		$\alpha = 0$		$\alpha = .20$		$\alpha = .50$		$\alpha = .90$		$\alpha = .99$	
		$\widehat{p}$	$\widehat{l}$ $\times 10^{-2}$	$\widehat{p}$	$\widehat{l}$ $\times 10^{-1}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$
First-order asymptotic	(CI_1)	1.000	1.163	.823	.548	.909	.249	.932	.424	.922	.265
Efron's percentile (BC)	(CI_S)	.948	2.587	.945	.670	.950	.253	.952	.430	.941	.315
Hall's percentile	(CI_H)	1.000	2.511	.680	.630	.821	.236	.867	.429	.916	.301
Studentized	(CI_{SH})	1.000	1.329	.764	.724	.958	.311	.924	.461	.935	.326
Superefficient est.	(CI_5)	.946	.396	.635	.322	.909	.248	.932	.424	.922	.265
Subsampling	(CI_6)	1.000	7.258	.929	1.778	.987	.298	.942	.439	.936	.270
Subsampling with $\widehat{\tau}_T$	(CI_7)	1.000	2.489	.827	.533	.790	.203	.844	.411	.929	.348
Indirect	(CI_8)	.956	2.753	.954	.715	.953	.202	.949	.451	.953	.304

Table 4.6: Impulse response of order $h = 4$ ($\phi_4 = \alpha^4$), estimated coverage probability ($\widehat{p}$) and average length ($\widehat{l}$) of estimated 95% CIs, sample size $T = 100$.

		$\alpha = 0$		$\alpha = .20$		$\alpha = .50$		$\alpha = .90$		$\alpha = .99$	
		$\widehat{p}$	$\widehat{l}$ $\times 10^{-2}$	$\widehat{p}$	$\widehat{l}$ $\times 10^{-1}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$	$\widehat{p}$	$\widehat{l}$
First-order asymptotic	(CI_1)	1.000	.243	.778	.199	.888	.171	.923	.497	.917	.340
Efron's percentile (BC)	(CI_S)	.000	.747	.971	.298	.950	.185	.952	.503	.941	.398
Hall's percentile	(CI_H)	.977	.711	.620	.273	.785	.167	.856	.486	.902	.447
Studentized	(CI_{SH})	.979	.361	.764	.329	.962	.236	.929	.566	.937	.436
Superefficient est.	(CI_5)	.946	.065	.564	.124	.888	.171	.923	.497	.917	.340
Subsampling	(CI_6)	1.000	13.047	.948	2.449	.991	.233	.943	.537	.941	.359
Subsampling with $\widehat{\tau}_T$	(CI_7)	1.000	.489	.581	.198	.733	.135	.820	.454	.915	.422
Indirect	(CI_8)	.951	.751	.954	.321	.953	.202	.949	.540	.953	.397

reductions of the length may be possible by taking full advantage of the actual asymptotic distribution. In this respect the CI_1 intervals are just outperformed by the CIs based on the superefficient estimator which is overall clearly the best method for the case $\alpha = 0$. However, for nonzero α both the CIs based on first–order asymptotics and on the superefficient estimator are problematic in the currently considered small sample context because their actual coverage levels deteriorate substantially for $\alpha \neq 0$. This is true in particular for the latter CIs. For instance, the coverage frequency of CI_5 is only 56.4% for $\alpha = 0.2$ when confidence intervals for α^4 are considered. Increasing the sample size to $T = 1,000$ in this case led to a far better coverage of 90.8% (results are not reported here).

Efron's percentile confidence intervals (BC) show a clear drawback for the case $\alpha = 0$ and even h (see Table 4.4 and Table 4.6). In these cases a coverage of 0% is observed. Hence CI_S has to be used cautiously if nothing is known about the actual parameter values. For $\alpha \neq 0$, CI_S shows excellent coverage. The latter finding is in line with other studies, e.g., Kilian (1998a), Kilian (1998b).

Hall's percentile method leads to similar results as the first–order asymptotic theory. In the majority of cases, the performance of the bootstrap intervals is even slightly worse than that of CI_1 with respect to both the coverage probability and the length of the intervals. The coverage probability deteriorates again for intermediate values of α and $h > 1$. This seems to be a small sample problem, however. Further simulation experiments showed that the theoretical behavior of CI_H is better for samples of size $T = 1,000$. The studentized version of Hall's bootstrap (CI_{SH}) leads to an almost overall better coverage than CI_H. However, as for CI_H its coverage deteriorates for intermediate values of α and $h > 1$.

The CIs based on the subsampling bootstrap (CI_6) produce roughly the

correct coverage level. In some cases their length is considerably greater than that of other methods, especially for $h > 2$ and α less or equal to 0.2. In addition, taking into account that for more complicated situations the subsampling bootstrap involves a substantially larger computational burden, the virtue of this method is difficult to see at least in the present context.

In the case of $\alpha = 0$, the subsampling procedure which additionally estimates the rate of convergence (CI_7) is superior to the other subsampling method for $h > 1$. Even though it estimates more conservative CIs it seems to have a slight advantage over the standard bootstrap in terms of interval length. This observation changes for $\alpha \neq 0$. The interval length is still shorter compared to the other subsampling method but it is achieved at the cost of reduced coverage probability. A reason for that might be a poor estimator of δ which implies poor estimation results for τ_T (see Table 4.2).

4.4 Summary and recommendations

The simulations show that all CIs have drawbacks. Since some of them work very poorly for the just considered simplest case there is clearly not much hope that they generally behave well in more complicated situations where higher order or higher dimensional processes are of interest unless the problematic parts in the parameter space are avoided.

The intervals CI_1, CI_S, CI_H, and CI_{SH}, and CI_7 allow a straightforward generalization to the case of multivariate autoregressive processes of higher order. However, CI_1, CI_S, CI_H, and CI_{SH} are asymptotically incorrect in particular cases of interest, which are mimicked by $\alpha = 0$ and $h > 0$ in the simplified context presented here. The subsampling confidence interval CI_6 can also be generalized easily if one finds a norming factor leading to a statistic with a nondegenerate limit distribution. Even the indirect method, which was

clearly the winner in the experiment, can be generalized in principle. However, this will certainly lead to quite substantial computational problems and, hence, this method will suffer from the "curse of dimensionality".

It is recommended in the literature to use biased corrected bootstrap confidence intervals (Efron and Tibshirani (1993), Kilian (1998b)). The effect of no or poor bias correction can be seen in the simulation study where CI_S failed completely to include the true parameter in the interesting case $\alpha = 0$ for even $h > 1$. The Hall percentile interval CI_H accounts implicitly for the bias. Therefore, this interval and refinements based on it, i.e. CI_{SH}, are recommended here in addition to a modeling strategy that avoids the problematic parts in the parameter space.

Since the problems of the standard methods are essentially caused by zero elements in the autoregressive matrices it may be a possible strategy to pretest for zero coefficients and specify subset VAR models. If the zero elements are specified correctly, all of the standard methods should work asymptotically. Thus, it may be worth spending some effort in model specification before an impulse response analysis is carried out.

This modeling strategy had been used in Benkwitz et al. (2001). In addition, the intervals CI_S to CI_{SH} and an iterated version of CI_H were computed for impulse responses. It turned out that the statistical interpretation of the results was not affected by the choice of the confidence interval. Therefore, Benkwitz et al. (2001) continued to use CI_H because of the theoretical advantage and the speed advantage compared to CI_{SH} and the iterated version.

Bibliography

Anderson, T.W. (1959). On asymptotic distribution of estimates of parameters of stochastic difference equations. *The Annals of Mathematical Statistics*, 30:676–687.

Andrews, D.W.K. (1993). Exactly median-unbiased estimation of first order autoregressive/unit root models. *Econometrica*, 61:139–165.

Aptech (1996). *Gauss. Systems and Graphics Manual*. Aptech Systems, Maple Valley.

Benkwitz, A. and Lütkepohl, H. and Neumann, M.H. (2000). Problems Related to Confidence intervals for impulse responses of autoregressive processes. *Econometric Reviews*, 19(1):69–103.

Benkwitz, A. and Lütkepohl, H. and Wolters, J. (2001). Comparison of Bootstrap confidence intervals for impulse responses of German monetary systems. *Macroeconomic Dynamics*, 5:81–100.

Bertail, P. and Politis, D.N. and Romano, J.P. (1999). On subsampling estimators with unknown rate of convergence. *Journal of the American Statistical Association*.

Bickel, P.J. and Götze, F. and van Zwet, W.R. (1997). Resampling fewer than n obserations: Gains, losses, and remedies for losses. *Statistica Sinica*, 7:1–31.

Breslaw, J. (2002). *Mercury. Interface Tools for GAUSS*. Econotron Software, Inc. Also at URL |http://www.econotron.com/mercury4/download/—.

Brüggemann, I. and Wolters, J. (1998). Money and Prices in Germany: Empirical Results for 1962 to 1996. In Galata, R. and Küchenhoff, H., editors, *Econometrics in Theory and Practice*. Physica, Heidelberg.

Brüggemann, R. and Lütkepohl, H. (2001). Lag Selection in Subset VAR Models with an Application to a U.S. Monetary System. In Friedmann, R., Knüppel, L., and Lütkepohl, H., editors, *A Festschrift in Honour of Joachim Frohn*, pages 107–128. LIT Verlag, Münster.

Budimlic, Zoran and Kennedy, Ken and Piper, Jeff (1999). The Cost of Being Object-Oriented: A Preliminary Study. *Scientific Computing*, 7(2):87–95.

Cribari-Neto, Francisco (1999). C for Econometricians. *Computational Economics*, 14(1):135–149.

Datta, S. (1996). On asymptotic properties of the bootstrap for AR(1) processes. *Journal of Statistical Planning and Inference*, 53:361–473.

Datta, S. and Sriram, T.N. (1997). A modified bootstrap for autoregression without stationarity. *Journal of Statistical Planning and Inference*, 59:19–30.

Davison, A.C. and Hinkley, D.V. (1997). *Bootstrap methods and their application*. Cambridge University Press, Cambridge.

de Wet, T. and van Wyk, J.W.J. (1986). Bootstrap confidence intervals for regression coefficients when the residuals are dependent. *Journal of Statistical Computation and Simulation*, 23:317–327.

Dhrymes, P.J. (1974). *Econometrics. Statistical Foundations and Applications*. Springer, New York. 2nd printing.

Doornik, J.A. and Hendry, D.F. (1997). *Modelling Dynamic Systems Unsing PcFiml 9.0 for Windows®*. International Thomson Business Press.

R. Eckstein and M. Loy and D. Wood (1998). *Java Swing*. O'Reilly.

Efron (1979). Bootstrap methods: another look at the jackknife. *Ann. Statist.*, 7:1–26.

Efron, B. and Tibshirani, R.J. (1986). Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical Science*, 1:54–77.

Efron, B. and Tibshirani, R.J. (1993). *An Introduction to the Bootstrap*. Chapman & Hall, New York.

Ehrmann, M. and Ellison, M. and Valla, Natacha (2001). Regime-dependent impulse response functions in a Markov-switching vector autoregressive model. Technical Report 11, Bank of Finland.

Fachin, S. and Bravetti, L. (1996). Asymptotic normal and bootstrap inference in structural VAR analysis. *Journal of Forecasting*, 15:329–341.

Goldberger, A.S. (1991). *A Course in Econometrics*. Harvard University Press, Cambridge.

Gosling, J. and McGilton, H. (2002). *The Java Language Environment*. Online at: http://java.sun.com/docs/white/.

Green, T.R.G. and Blackwell, A.F. (1996). Thinking about Visual Programs. In *Thinking with Diagrams*.

Griffiths, W. and Lütkepohl, H. (1989). Confidence intervals for impulse responses from VAR models: A comparison of asymptotic theory and simulation approaches. Discussion Paper, University of New England, Armidale, Australia.

Haase, K. and Lütkepohl, H. and Claessen, H. and Moryson, M. and Schneider, W. (1992). MulTi. A Menu–Driven GAUSS Program for Multiple Time Series Analysis. Institut für Statistik und Ökonometrie, Universität Kiel, Kiel.

Hall, P. (1988). Theoretical comparison of bootstrap confidence intervals (with discussion). *The Annals of Statistics*, 16:927–953.

Hall, P. (1992). *The Bootstrap and Edgeworth Expansion*. Springer, New York.

Hall, P. (1994). Methodology and theory for the bootstrap. In Engle, R. and McFadden, D., editors, *Handbook of Econometrics*, volume IV, chapter 39, pages 2341–2381. Elsevier Science.

Hamilton, J.D. (1994). *Time Series Analysis*. Princeton University Press, Princeton, New Jersey.

Heimann, G. and Kreiss, J.-P. (1996). Bootstrapping general first order autoregression. *Statistics and Probability Letters*, 30:87–98.

Heiss, J.H. (1999). JavaHelp™Technology to the Rescue. Online at http://java.sun.com/features/1999/04/javahelp.html.

IBM (1999). *Visual Composition*. Manual.

Johansen, Søren (1995). *Likelihood-Based Inference in Cointegrated Vector Autoregressive Models*. Oxford University Press, Oxford.

Judge, George G. and Hill, R. Carter and Griffiths, William E. and Lütkepohl, Helmut and Lee, Tsoung-Chao (1988). *Introduction to the Theory and Practice of Econometrics*. John Wiley & Sons, New York, 2nd edition.

Kilian, L. (1998a). Confidence intervals for impulse responses under departures from normality. *Econometric Reviews*, 17:1–29.

Kilian, L. (1998b). Small-sample confidence intervals for impulse response functions. *Review of Economics and Statistics*, 80:218–230.

Lehmann, E.L. (1999). *Elements of Large–Sample Theory*. Springer, New York.

Li, H. and Maddala, G.S. (1996). Bootstrapping time series models. *Econometric Reviews*, 15(2):115–158.

Liu, L.-M. and Chan, K.-K. and Montgomery, A.L. and Muller, M.E. (1995). A system-independent graphical user interface for statistical software. *Computational Statistics & Data Analysis*, 19:23–44.

Lütkepohl, H. (1993a). *Introduction to Multiple Time Series Analysis*. Springer-Verlag, Berlin, 2nd edition.

Lütkepohl, H. (1993b). MulTi - A Menu-Driven GAUSS Program for Multiple Time Series Analysis. *Computational Statistics*, 8:161–163.

Lütkepohl, H. (2000). Bootstrapping impulse responses in VAR analyses. In *Compstat, Proceedings in Computational Statistics, 14th Symposium held in Utrecht, The Netherlands*, pages 109–119. Physica, Heidelberg.

Lütkepohl, Helmut and Breitung, J. (1997). Impulse Response Analysis of Vector Autoregressive Processes. In Heij, C., Schumacher, J., Hanzon, B., and Praagman, C., editors, *System Dynamics in Economic and Financial Models*, pages 299–326. John Wiley and Sons Ltd, New York.

Lütkepohl, H. and Müller, C. and Saikkonen, P. (2001). Unit root tests for time series with a structural break when the break point is known. In Hsiao,

Morimune, and Powell, editors, *Nonlinear Statistical Modeling, Proceedings of the 13. International Symposium in Economic Theory and Economics Essays in Honor of Takeshi Amemiya*, pages 327–348. Cambridge University Press.

Lütkepohl, H. and Poskitt, D.S. (1996). Testing for causation using infinite order vector autoregressive processes. *Econometric Theory*, 12:61–87.

Lütkepohl, H. and Wolters, J. (1998). A money demand system for German M3. *Empirical Economics*, 23:371–386.

Politis, D.N. and Romano, J.P. (1994). Large sample confidence regions based on subsamples under minimal assumptions. *The Annals of Statistics*, 22(4):2031–2050.

Runkle, D.E. (1987). Vector Autoregresions and Reality. *Journal of Business & Economic Statistics*, 5(4):437–442.

Saikkonen, P. and Lütkepohl, H. (2000). Asymptotic inference on nonlinear functions of the coefficients of infinite order cointegrated VAR processes. In Barnett, A., Hendry, D., Hylleberg, S., Teräsvirta, T., Tjostheim, D., and Würtz, A., editors, *Nonlinear Econometric Modelling in Time Series Analysis*, pages 165–201. Cambridge University Press, Cambridge.

M. Schader and L. Schmidt-Thieme (1996). *Java. Einführung in die objektorientierte Programmierung.* Springer.

Schiffer, S. (1998). *Visuelle Programmierung - Grundlagen und Einsatzmöglichkeiten.* Addison-Wesley-Longman.

Sims, C.A. (1980). Macroeconomics and Reality. *Econometrica*, 48:1–48.

Sims, C.A. and Zha, T. (1999). Error bands for impulse responses. *Econometrica*, 67(5):1113–1155.

Sun (2001). The Java Tutorial. Online at http://java.sun.com/docs/books/tutorial/index.html.

Sun (2002). Java™Technology: Products & APIs. Online at http://java.sun.com/products/.

Teräsvirta, T. (1998). *Handbook of Applied Economic Statistics*, chapter 15, Modeling Economic Relationships with STR. Marcel Dekker.

Tschernig, Rolf and Yang, Lijian (2000). Nonparametric Lag Selection for Time Series. *Journal of Time Series Analysis*, 21(4):457 – 487.

Wampler, B.E. (2001). The Essence of Object-Oriented Programming with Java and UML. Online at URL: http://www.objectcentral.com/oobook/oobook.htm.

Watson, M.W. (1987). Comment on Runkle (1987). *Journal of Business & Economic Statistics*, 5(4):451–453.

Yamamoto, Y. and Nakano, J. and Fujiwara, T. and Kobayshi, I. (2001). A Mixed User Interface for a Statistical System. Discussion Paper No. 77, SFB 373, Humboldt-Universität zu Berlin.

Appendix A

License agreement

JStatCom (Chapter 2), *Gauss Control* (Appendix C), and JMulTi (Chapter 3) are research, teaching and learning software that can be used free of charge. They do not come with any support. Note the disclaimer at the end of the agreement. They are offered to the public in the spirit of the University of Illinois/NCSA Open Source License:

Developed by:
Alexander Benkwitz, Markus Krätzig
Humboldt University Berlin
http://ise.wiwi.hu-berlin.de

cense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

3. Neither the names of Alexander Benkwitz, Markus Krätzig, Humboldt University Berlin, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

(Source: http://www.opensource.org/licenses/UoI-NCSA.html)

Appendix B

Documentation of Java library JStatCom

B.1 The Java library JStatCom

The classes in the Java library JStatCom are organized in the packages `gauss`, `gauss.control`, `util`, `util.component`, and `xlm`. The packages `util` and `util.component` contain classes that were provided by other people in the world wide web:

`util.PrintfFormat` a class that allows the formatting of an array of objects embedded within a string, see http://developer.java.sun.com/developer/technicalArticles/Programming/sprintf/

`util.SwingWorker` a class that performs GUI-related work in a dedicated thread, see http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html

`util.component.CardPanel` a simpler alternative to a JPanel with a CardLayout, see http://java.sun.com/products/jfc/tsc/articles/cardpanel/

`util.component.MultiLineLabelUI` a rendering class that enables multi-lines in a JLabel see http://codeguru.earthweb.com/java/articles/198.shtml

The classes are documented in the Java source code. Although it is recognized to have some documentation at hand, it is not reported here. Reporting the full documentation would go beyond the scope of this work.

B.2 A simple GUI example

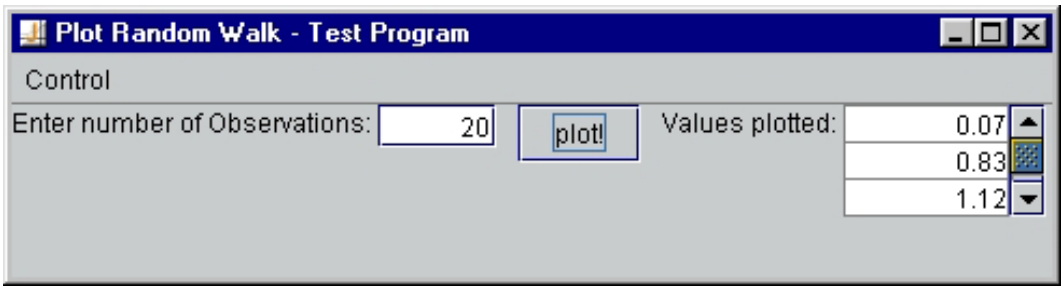


Figure B.1: Screen shot of sample application

The simple GUI application shown in Figure B.1 is explained line by line. Note that all instructions are placed in the `main()` method of the class `RandomWalk`. This is done for convenience reasons but is not appropriate when writing more complex programs.

Import statements of packages used by RandomWalk class:

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import gauss.*;
5  import gauss.control.*;
6  import util.component.*;
```

Class declaration, begin of main-method body. The code is included in a `try`-block for catching and reporting exceptions:

```
7  class RandomWalk {
8   public static void main(String[] args) {
9    try {
```

Creating a container that holds all other components:

```
10     final GaussPanel gaussPanel = new GaussPanel();
11     BoxLayout layout =
12               new BoxLayout(gaussPanel, BoxLayout.X_AXIS);
13     gaussPanel.setLayout(layout);
```

Creating a label that names the input component. The input component writes valid input to the database. The data is identified by the name T:

```
14     JLabel label = new JLabel("Enter number of Observations: ");
15     label.setAlignmentY(JComponent.TOP_ALIGNMENT);
16     gaussPanel.add(label);
17
18     GaussDataTextField textField = new GaussDataTextField(5);
19     textField.setFixedSize(true);
20     textField.setSymbolName("T");
21     textField.setLowerBound(1.0);
22     textField.setDataType(Verifier.INTEGER);
23     textField.setInitialNumber(20.0);
24     textField.setAlignmentY(JComponent.TOP_ALIGNMENT);
25     gaussPanel.add(textField);
```

Add some space between the input component and the following component:

```
26     gaussPanel.add(
27                 Box.createRigidArea(new Dimension(10, 0)));
```

Create a button that triggers the Gauss procedure call. The procedure call loads the Gauss library pgraph (if necessary), writes T to Gauss, computes $y_t = \sum_{i=1}^{T} x_i$, $x_i \sim N(0,1)$, plots the y_t, puts the resulting graphic window to the front, and reads y_t from Gauss:

```
28     JButton button = new JButton("plot!");
29     button.addActionListener(new ActionListener() {
30      public void actionPerformed(ActionEvent e) {
31       gaussPanel.gaussStart("Compute and plot random walk");
```

```
32      gaussPanel.gaussLoadLibrary("pgraph");
33      gaussPanel.gaussWrite("T");
34      gaussPanel.gaussExec("y = cumsumc(rndn(T,1))");
35      gaussPanel.gaussExec("xy(seqa(1,1,T),y)");
36      gaussPanel.gaussExec("dllcall showLastGraphic");
37      gaussPanel.gaussRead("y");
38      gaussPanel.gaussRun();
39     }
40    });
41    button.setAlignmentY(JComponent.TOP_ALIGNMENT);
42    gaussPanel.add(button);
```

Add some space between the input component and the following component:

```
43    gaussPanel.add(
44                Box.createRigidArea(new Dimension(10, 0)));
```

Creating a label that names the output component:

```
45    label = new JLabel("Values plotted: ");
46    label.setAlignmentY(JComponent.TOP_ALIGNMENT);
47    gaussPanel.add(label);
```

The output component `table` displays data from the database. The data is identified by the symbol name y. Since `table` may contain to many rows or columns for a computer screen it is embedded in the `scrollpane`. The `scrollpane` displays only a pre-defined number of rows and columns of `table`. By specifying a minimum and maximum number the size can float within this range depending on the data and space available:

```
48    GaussDataTable table = new GaussDataTable();
49    table.setSymbolName("y");
50    table.setEditable(false);
51    table.setMinimumColumnWidth(75);
52    table.setDynamicColumnWidth(true);
53    GaussDataTableScrollPane scrollPane =
54                               new GaussDataTableScrollPane(table);
55    scrollPane.setMinimumVisibleRows(1);
56    scrollPane.setMaximumVisibleRows(15);
57    scrollPane.setMinimumVisibleColumns(1);
58    scrollPane.setMaximumVisibleColumns(1);
```

```
59     scrollPane.setAlignmentY(JComponent.TOP_ALIGNMENT);
60     gaussPanel.add(scrollPane);
```

Creating a GaussFrame which establishes the connection to the software package Gauss. The show-command brings up the `frame` on the screen:

```
61     GaussFrame frame =
62       new GaussFrame("Plot Random Walk - Test Program");
63     frame.setSystemExitWhenClosing(true);
64     frame.getGauss().startGauss();
65     frame.setJMenuBar(new JMenuBar());
66     frame.getJMenuBar().add(new GaussMenu());
67     frame.setContentPane(gaussPanel);
68     frame.setSize(400, 300);
69     frame.show();
```

Ends the try-block, the body of the main method and the body of the class.

```
70    } catch (Throwable throwable) {
71     throwable.printStackTrace();
72     System.exit(1);
73    } // end of catch()
74   } // end of main()
75 } // end of class
```

B.3 Package util

B.3.1 Package Contents

Interfaces

Classes

B.3.2 Interfaces

Interface ProcedureCallConstants

A collection of constants generally used for executing calls to an external statistical software package.

Declaration

public interface ProcedureCallConstants

Fields

public static final int EXECUTE
public static final int WRITE
public static final int WRITEARRAY
public static final int READ
public static final int READARRAY
public static final int READ_LOCAL
public static final int READARRAY_LOCAL
public static final int SHOW
public static final int LOADLIBRARY
public static final int LOADDLIBRARY

Interface ProcedureCallListener

The listener interface for receiving ProcedureCallEvents. The class that is interested in processing a ProcedureCallEvent implements this interface. The object created with that class is then registered with a ProcedureCall using the object's addProcedureCallListener method. When an ProcedureCallEvent occurs, the specific listener object's listener method is invoked.

Declaration

public interface ProcedureCallListener **implements** java.util.EventListener

Methods

public void procedureCallFinished (`util.ProcedureCallEvent e`)

This method is called when a ProcedureCall object finishes.
Firering this event is the last action in the ProcedureCall run() method. Note, that a ProcedureCallFinished event does not necessarily mean that the procedure call was successful.

public void procedureCallProgress (`util.ProcedureCallEvent e`)
This method is called when the progress property of the ProcedureCall object changed.

public void procedureCallRegistered (`util.ProcedureCallEvent e`)
This method is called when a ProcedureCall object was successfully registered with an object that manages various ProcedureCall objects.
Registering a procedure call always proceeds starting it.

public void procedureCallStarted (`util.ProcedureCallEvent e`)
This method is called when a ProcedureCall object starts execution.
This means that the run() method of the thread ProcedureCall was just called by some managing object.

B.3.3 Classes

Class PrintfFormat

PrintfFormat allows the formatting of an array of objects embedded within a string. Primitive types must be passed using wrapper types. The formatting is controlled by a control string.
A control string is a Java string that contains a control specification. The control specification starts at the first percent sign (%) in the string, provided that this percent sign

1. is not escaped protected by a matching % or is not an escape % character,
2. is not at the end of the format string, and
3. precedes a sequence of characters that parses as a valid control specification.

A control specification usually takes the form:

```
% ['-+ #0]* [0..9]* { . [0..9]* }+
                { [hlL] }+ [idfgGoxXeEcs]
```

There are variants of this basic form that are discussed below.
The format is composed of zero or more directives defined as follows:

- ordinary characters, which are simply copied to the output stream;
- escape sequences, which represent non-graphic characters; and
- conversion specifications, each of which results in the fetching of zero or more arguments.

The results are undefined if there are insufficient arguments for the format. Usually an unchecked exception will be thrown. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored. In format

strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.
Conversions can be applied to the `n`th argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion characer % is replaced by the sequence %`n`$, where `n` is a decimal integer giving the position of the argument in the argument list.
In format strings containing the %`n`$ form of conversion specifications, each argument in the argument list is used exactly once.

Escape Sequences

The following table lists escape sequences and associated actions on display devices capable of the action.

Sequence	**Name**	**Description**
‘‘	backlash	None.
‘a	alert	Attempts to alert the user through audible or visible notification.
‘b	backspace	Moves the printing position to one column before the current position, unless the current position is the start of a line.
‘f	form-feed	Moves the printing position to the initial printing position of the next logical page.
‘n	newline	Moves the printing position to the start of the next line.
‘r	carriage-return	Moves the printing position to the start of the current line.
‘t	tab	Moves the printing position to the next implementation- defined horizontal tab position.
‘v	vertical-tab	Moves the printing position to the start of the next implementation-defined vertical tab position.

Conversion Specifications

Each conversion specification is introduced by the percent sign character (%). After the character %, the following appear in sequence:
Zero or more flags (in any order), which modify the meaning of the conversion specification.
An optional minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces by default on the left; t will be padded on the right, if the left- adjustment flag (-), described below, is given to the field width. The field width takes the form of a decimal integer. If the conversion character is s, the field width is the the minimum number of characters to be printed.
An optional precision that gives the minumum number of digits to appear for the d, i, o, x or X conversions (the field is padded with leading zeros); the number of digits to appear after the radix character for the e, E, and f conversions, the maximum number of significant digits for the g and G conversions; or the maximum number

of characters to be written from a string is s and S conversions. The precision takes the form of an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with a c conversion character the precision is ignored.
An optional h specifies that a following d, i, o, x, or X conversion character applies to a type short argument (the argument will be promoted according to the integral promotions and its value converted to type short before printing).
An optional l (ell) specifies that a following d, i, o, x, or X conversion character applies to a type long argument.
A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer argument supplised the field width precision. The argument that is actually converted is not fetched until the conversion letter is seen, so the the arguments specifying field width or precision must appear before the argument (if any) to be converted. If the precision argument is negative, it will be changed to zero. A negative field width argument is taken as a - flag, followed by a positive field width.
In format strings containing the %`n`$ form of a conversion specification, a field width or precision may be indicated by the sequence *`m`$, where m is a decimal integer giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision.
The format can contain either numbered argument specifications (that is, %`n`$ and *`m`$), or unnumbered argument specifications (that is % and *), but normally not both. The only exception to this is that %% can be mixed with the %`n`$ form. The results of mixing numbered and unnumbered argument specifications in a format string are undefined.

Flag Characters

The flags and their meanings are:

' integer portion of the result of a decimal conversion (%i, %d, %f, %g, or %G) will be formatted with thousands' grouping characters. For other conversions the flag is ignored. The non-monetary grouping character is used.

\- result of the conversion is left-justified within the field. (It will be right-justified if this flag is not specified).

\+ result of a signed conversion always begins with a sign (+ or -). (It will begin with a sign only when a negative value is converted if this flag is not specified.)

<space> If the first character of a signed conversion is not a sign, a space character will be placed before the result. This means that if the space character and + flags both appear, the space flag will be ignored.

\# value is to be converted to an alternative form. For c, d, i, and s conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x or X conversion, a non-zero result has 0x or 0X prefixed to it, respectively. For e, E, f, g, and G conversions, the result always contains a radix character, even if no digits follow the radix character (normally, a decimal point appears in the result of these conversions

only if a digit follows it). For g and G conversions, trailing zeros will not be removed from the result as they normally are.

0 d, i, o, x, X, e, E, f, g, and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, x, and X conversions, if a precision is specified, the 0 flag will be ignored. For c conversions, the flag is ignored.

Conversion Characters

Each conversion character results in fetching zero or more arguments. The results are undefined if there are insufficient arguments for the format. Usually, an unchecked exception will be thrown. If the format is exhausted while arguments remain, the excess arguments are ignored.
The conversion characters and their meanings are:

d,i The int argument is converted to a signed decimal in the style [-]dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

o The int argument is converted to unsigned octal format in the style ddddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

x The int argument is converted to unsigned hexadecimal format in the style dddd; the letters abcdef are used. The precision specifies the minimum numberof digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

X Behaves the same as the x conversion character except that letters ABCDEF are used instead of abcdef.

f The floating point number argument is written in decimal notation in the style [-]ddd.ddd, where the number of digits after the radix character (shown here as a decimal point) is equal to the precision specification. A Locale is used to determine the radix character to use in this format. If the precision is omitted from the argument, six digits are written after the radix character; if the precision is explicitly 0 and the # flag is not specified, no radix character appears. If a radix character appears, at least 1 digit appears before it. The value is rounded to the appropriate number of digits.

e,E The floating point number argument is written in the style [-]d.ddde{+-}dd (the symbols {+-} indicate either a plus or minus sign), where there is one digit

before the radix character (shown here as a decimal point) and the number of digits after it is equal to the precision. A Locale is used to determine the radix character to use in this format. When the precision is missing, six digits are written after the radix character; if the precision is 0 and the # flag is not specified, no radix character appears. The E conversion will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. However, if the value to be written requires an exponent greater than two digits, additional exponent digits are written as necessary. The value is rounded to the appropriate number of digits.

g,G The floating point number argument is written in style f or e (or in sytle E in the case of a G conversion character), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted: style e (or E) will be used only if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.

c,C The integer argument is converted to a char and the result is written.

s,S The argument is taken to be a string and bytes from the string are written until the end of the string or the number of bytes indicated by the precision specification of the argument is reached. If the precision is omitted from the argument, it is taken to be infinite, so all characters up to the end of the string are written.

% Write a % character; no argument is converted.

If a conversion specification does not match one of the above forms, an IllegalArgumentException is thrown and the instance of PrintfFormat is not created.
If a floating point value is the internal representation for infinity, the output is [+]Infinity, where Infinity is either Infinity or Inf, depending on the desired output string length. Printing of the sign follows the rules described above.
If a floating point value is the internal representation for "not-a-number,"the output is [+]NaN. Printing of the sign follows the rules described above.
In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.
The behavior is like printf. One exception is that the minimum number of exponent digits is 3 instead of 2 for e and E formats when the optional L is used before the e, E, g, or G conversion character. The optional L does not imply conversion to a long long double.
The biggest divergence from the C printf specification is in the use of 16 bit characters. This allows the handling of characters beyond the small ASCII character set and allows the utility to interoperate correctly with the rest of the Java runtime environment.

Omissions from the C printf specification are numerous. All the known omissions are present because Java never uses bytes to represent characters and does not have pointers:

- %c is the same as %C.
- %s is the same as %S.
- u, p, and n conversion characters.
- %ws format.
- h modifier applied to an n conversion character.
- l (ell) modifier applied to the c, n, or s conversion characters.
- ll (ell ell) modifier to d, i, o, u, x, or X conversion characters.
- ll (ell ell) modifier to an n conversion character.
- c, C, d,i,o,u,x, and X conversion characters apply to Byte, Character, Short, Integer, Long types.
- f, e, E, g, and G conversion characters apply to Float and Double types.
- s and S conversion characters apply to String types.
- All other reference types can be formatted using the s or S conversion characters only.

Most of this specification is quoted from the Unix man page for the sprintf utility.

Declaration

public class PrintfFormat **extends** java.lang.Object

Constructors

public PrintfFormat (`java.util.Locale` **locale**,
`java.lang.String` **fmtArg**)

Constructs an array of control specifications possibly preceded, separated, or followed by ordinary strings. Control strings begin with unpaired percent signs. A pair of successive percent signs designates a single percent sign in the format.

public PrintfFormat (`java.lang.String` **fmtArg**)

Constructs an array of control specifications possibly preceded, separated, or followed by ordinary strings. Control strings begin with unpaired percent signs. A pair of successive percent signs designates a single percent sign in the format.

Methods

public String sprintf( )
Format nothing. Just use the control string.

public String sprintf (`double` **x**)
Format a double.

public String sprintf (`int` **x**)
Format an int.

public String sprintf (`long` **x**)
Format an long.

public String sprintf (`java.lang.Object` **x**)
Format an Object. Convert wrapper types to their primitive equivalents and call the appropriate internal formatting method. Convert Strings using an internal formatting method for Strings. Otherwise use the default formatter (use toString).

public String sprintf (`java.lang.Object[]` **o**)
Format an array of objects. Byte, Short, Integer, Long, Float, Double, and Character arguments are treated as wrappers for primitive types.

public String sprintf (`java.lang.String` **x**)
Format a String.

Class **ProcedureCall**

Thread that executes communication with and computation requests to some statistical software.
The task should be programmed in the `mainCode()`, and `postCode()` methods. The `run()` method is made final and executes the `mainCode()`, and `postCode()` methods in a predetermined way.

Declaration

public abstract class ProcedureCall **extends** java.lang.Thread
implements java.io.Serializable,
ProcedureCallConstants,
java.lang.Cloneable

Constructors

public ProcedureCall( )

Methods

public void addProcedureCallListener (`util.ProcedureCallListener` **l**)
Registers listener *l* so that it will receive ProcedureCallEvents.

public void fireProcedureCallRegistered( )
Execute this method when a ProcedureCall managing object receives an instance of this class.

public double getProgress( )
Returns estimated progress.

public long getStartTime( )
Returns the system time at thread start. This time is filed when the `start()` method is executed and is the value of `System.currentTimeMillis()`.

public boolean isSuccess( )
Returns whether the thread is/was executed successful.

public void removeProcedureCallListener (`util.ProcedureCallListener` **l**)

Unregisters listener *l* so that it will no longer receive ProcedureCallEvents.

public final void run( )
This method cannot be overwritten. Place the code instead in the `mainCode()`, and `postCode()` methods.

public void setProgress (`double` **newProgress**)
Set a new progress coefficient.

public void start( )
Starts the ProcedureCall thread, writes a time stamp to the `startTime`-field and notifies all interested listeners about the start.

Class **ProcedureCallAdapter**

An abstract adapter class for receiving procedure call events. The methods in this class are empty. This class exists as convenience for creating listener objects.

Declaration

public abstract class ProcedureCallAdapter **extends** java.lang.Object
implements ProcedureCallListener

Constructors

public ProcedureCallAdapter( )

Methods

public void procedureCallFinished (`util.ProcedureCallEvent` **e**)

public void procedureCallProgress (`util.ProcedureCallEvent` **e**)

public void procedureCallRegistered (`util.ProcedureCallEvent` **e**)

public void procedureCallStarted (`util.ProcedureCallEvent` **e**)

Class **ProcedureCallEvent**

An event which indicates that a procedure call is processed.
This event is generated by a ProcedureCall object when:

- it gets registered
- execution starts
- progress in execution is reported
- it finishes

A ProcedureCallEvent object is passed to every ProcedureCallListener or ProcedureCallAdapter object which registered to receive the procedure call events using the ProcedureCall's addProcedureCallListener method.

Declaration

public class ProcedureCallEvent **extends** java.util.EventObject

Fields

public static final int CALL_REGISTERED
public static final int CALL_STARTED
public static final int CALL_FINISHED
public static final int CALL_PROGRESS

Constructors

public ProcedureCallEvent (`util.ProcedureCall` **source**,
`int` **type**)

Constructs a ProcedureCallEvent object with the specified source object, and type.

Methods

public ProcedureCall getProcedureCall( )
Returns the originator of the event.

public int getType( )
Returns the event type.
The event type is one of

- `CALL_REGISTERED`
- `CALL_STARTED`
- `CALL_FINISHED`
- `CALL_PROGRESS`

Class **SwingWorker**

This is the 3rd version of SwingWorker (also known as SwingWorker 3), an abstract class that you subclass to perform GUI-related work in a dedicated thread. For instructions on using this class, see:
http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html
Note that the API changed slightly in the 3rd version: You must now invoke start() on the SwingWorker after creating it.

Declaration

public abstract class SwingWorker **extends** java.lang.Object

Constructors

public SwingWorker( )
Start a thread that will call the `construct` method and then exit.

Methods

public abstract Object construct( )
Compute the value to be returned by the `get` method.

public void finished( )
Called on the event dispatching thread (not on the worker thread) after the `construct` method has returned.

public Object get( )
Return the value created by the `construct` method. Returns null if either the constructing thread or the current thread was interrupted before a value was produced.

public void interrupt( )
A new method that interrupts the worker thread. Call this method to force the worker to stop what it's doing.

public void start( )
Start the worker thread.

B.4 Package gauss.control

B.4.1 Package Contents

Classes

B.4.2 Classes

Class **DesktopTheme**

GUI theme for desktop mode.

Declaration

public class DesktopTheme
extends javax.swing.plaf.metal.DefaultMetalTheme

Constructors

public DesktopTheme( )

Methods

public FontUIResource getControlTextFont( )

public FontUIResource getMenuTextFont( )

public FontUIResource getSubTextFont( )

public FontUIResource getSystemTextFont( )

public FontUIResource getUserTextFont( )

public FontUIResource getWindowTextFont( )

public FontUIResource getWindowTitleFont( )

Class **GaussDataTreeNode**

Tree node for displaying GaussData in the GaussSymbolTableTree.

Declaration

public class GaussDataTreeNode
extends javax.swing.tree.DefaultMutableTreeNode

Constructors

public GaussDataTreeNode (`gauss.GaussData` **gaussData**)

public GaussDataTreeNode (`gauss.GaussData` **gaussData**,
`gauss.GaussSymbolTable` **newSymbolTable**)

METHODS

public String getDescription( )

CLASS **GaussMenu**

Pre-build menu for accessing the GaussSymbolTableTree and other standard tools.

DECLARATION

public class GaussMenu **extends** javax.swing.JMenu
implements java.awt.event.ActionListener

CONSTRUCTORS

public GaussMenu( )

METHODS

public void actionPerformed (`java.awt.event.ActionEvent` **e**)

CLASS **GaussSymbolTableTree**

Visual representation of all GaussSymbolTables within an application. This object provides the ability to navigate through all GaussSymbolTables and even manipulate the GaussData objects stored in them.

DECLARATION

public class GaussSymbolTableTree **extends** javax.swing.JTree

CONSTRUCTORS

public GaussSymbolTableTree( )

METHODS

public static void addGaussSymbolTableTreeNode
(`java.lang.Object` **node**)

public String convertValueToText (`java.lang.Object` **value**,
`boolean` **selected**,
`boolean` **expanded**,
`boolean` **leaf**,
`int` **row**,
`boolean` **hasFocus**)

public static void removeGaussSymbolTableTreeNode
(`java.lang.Object` **node**)

Class **PresentationTheme**

GUI theme for presentation mode. This is especially useful when using the Java application in classes.

Declaration

public class PresentationTheme
extends javax.swing.plaf.metal.DefaultMetalTheme

Constructors

public PresentationTheme( )

Methods

public FontUIResource getControlTextFont( )

public FontUIResource getMenuTextFont( )

public FontUIResource getSubTextFont( )

public FontUIResource getSystemTextFont( )

public FontUIResource getUserTextFont( )

public FontUIResource getWindowTextFont( )

public FontUIResource getWindowTitleFont( )

Class **ShowSymbolControlFrameAction**

Predefined action for showing the frame that contains the application wide GaussSymbolTableTree.

DECLARATION

public class ShowSymbolControlFrameAction
extends javax.swing.AbstractAction

CONSTRUCTORS

public ShowSymbolControlFrameAction( )

METHODS

public void actionPerformed (`java.awt.event.ActionEvent` **arg1**)

CLASS **ShowSymbolControlFrameMenuItem**

Predefined menu item for showing the frame that contains the application wide GaussSymbolTableTree.

DECLARATION

public class ShowSymbolControlFrameMenuItem
extends javax.swing.JMenuItem

CONSTRUCTORS

public ShowSymbolControlFrameMenuItem( )

CLASS **SymbolControl**

Container component for navigating through all application wide GaussDataTables. This component contains a GaussSymbolTableTree and components for displaying and manipulating the GaussData objects.

DECLARATION

public class SymbolControl **extends** gauss.GaussPanel

CONSTRUCTORS

public SymbolControl( )

Methods

public static void addGaussLocalSymbolRoot
(`gauss.GaussLocalSymbolRoot` **arg**)

public static void addSymbolTable
(`gauss.GaussSymbolTable` **gaussSymbolTable**)

public void setGauss (`gauss.Gauss` **g**)

public void setParent (`java.awt.Frame` **newParent**)

Class SymbolDescription

Class for displaying the symbol description in a specialized JDialog.

Declaration

public class SymbolDescription **extends** javax.swing.JDialog

Constructors

public SymbolDescription( )

public SymbolDescription (`java.awt.Frame` **owner**,
`java.lang.String` **title**,
`boolean` **modal**)

Methods

public void setDescription (`java.lang.String` **description**)

Class SymbolTableTreeCellRenderer

CellRenderer for the GaussSymbolTableTree.

Declaration

public class SymbolTableTreeCellRenderer
extends javax.swing.tree.DefaultTreeCellRenderer

Constructors

public SymbolTableTreeCellRenderer( )

Methods

public Component getTreeCellRendererComponent
(`javax.swing.JTree` **tree**,
`java.lang.Object` **value**,
`boolean` **sel**,
`boolean` **expanded**,
`boolean` **leaf**,
`int` **row**,
`boolean` **hasFocus**)

Class **SymbolTableTreeNode**

Definition of nodes in the SymbolTableTree. A node in the SymbolTableTree is defined to be a component that holds a GaussSymbolTable physically, i.e. instances of the classes GaussFrame, GaussInternalFrame, or LocalGaussPanel.

Declaration

public class SymbolTableTreeNode
extends javax.swing.tree.DefaultMutableTreeNode
implements java.beans.PropertyChangeListener,
javax.swing.event.TreeWillExpandListener

Constructors

public SymbolTableTreeNode (`gauss.GaussFrame` **gaussFrame**)

public SymbolTableTreeNode
(`gauss.GaussInternalFrame` **internalFrame**)

public SymbolTableTreeNode (`gauss.GaussLocalSymbolRoot` **local**)

public SymbolTableTreeNode (`gauss.LocalGaussPanel` **localGaussPanel**)

Methods

public String getTitleOfComponent( )

public void propertyChange (`java.beans.PropertyChangeEvent` **evt**)

public void sortChilds( )

public void treeWillCollapse (`javax.swing.event.TreeExpansionEvent` **e**)

public void treeWillExpand (`javax.swing.event.TreeExpansionEvent` **e**)

B.5 Package gauss

B.5.1 Package Contents

Interfaces

Classes

B.5.2 Interfaces

Interface **GaussEngineRoot**

All visual objects (coming from java.awt.Component or one of its sub classes) working with the GAUSS program or doing data manipulation should implement this interface.

Declaration

public interface GaussEngineRoot

Methods

public Gauss getGauss( )

The implementation should return reference to the Gauss-object that keeps contact with the Gauss program.

Interface **GaussLocalSymbolRoot**

Components implementing this interface should provide reference to a local GaussSymbolTable.

Declaration

public interface GaussLocalSymbolRoot

Methods

public GaussSymbolTable getLocalGaussSymbolTable( )

Components implementing this interface are defined to be the top level component for local GaussData requests. Every get/setLocalGaussData request of child components added to this component refer to the GaussSymbolTable returned by this method.

Interface **GaussSymbolRoot**

Components implementing this interface should provide reference to a non-local GaussSymbolTable.

Declaration

public interface GaussSymbolRoot

METHODS

public GaussSymbolTable getGaussSymbolTable( )
This method should return the contextual right GaussSymbolTable.

INTERFACE **GaussThreadListener**

Listener interface for GaussThread events.

DECLARATION

public interface GaussThreadListener **implements** java.util.EventListener

METHODS

public void gaussThreadFinished( )
Gets executed whenever a GaussThread finishes.

public void gaussThreadFinished (`int` **actionType**)
Gets executed whenever a GaussThread finishes with a certain action.

public void gaussThreadFinished (`int` **actionType**,
`java.lang.Object[]` **param**)
Gets executed whenever a GaussThread finishes with a certain action and output parameters.

B.5.3 Classes

CLASS **ElementChangeEvent**

Event to signal the change of a single element in the data of a GaussData object. An ElementChangeEvent gets delivered from a GaussData object whenever some interested object is registered as a PropertyChangeListener or VetoableChangeListener. This is advantageous since it first delivers the old value to the listener object and second the listener object may veto the change *before* the GaussData object changes.

DECLARATION

public class ElementChangeEvent **extends** java.beans.PropertyChangeEvent

Constructors

public ElementChangeEvent (`javax.swing.table.TableModel` **source**,
`double` **oldValue**,
`double` **newValue**,
`int` **row**,
`int` **column**)

Constructs an ElementChangeEvent object for numeric data. The primitive type double is wrapped to a Double-object. The `getNewValue()` and `getOldValue()` methods return Objects of type java.lang.Double if the event is constructed this way. `getPropertyName()` returns `GaussData.MATRIX_CHANGE`

public ElementChangeEvent (`javax.swing.table.TableModel` **source**,
`java.lang.String` **newValue**,
`int` **row**,
`int` **column**)

Constructs an ElementChangeEvent object for string data.
The `getNewValue()` and `getOldValue()` methods return Objects of type java.lang.String if the event is constructed this way. `getPropertyName()` returns `GaussData.STRING_CHANGE`

Methods

public int getColumn( )

Returns the column index of the changed element. The column index has a range from 0,...,cols(x)-1.

public int getRow( )

Returns the row index of the changed element. The row index has a range from 0,...,rows(x)-1.

Class **Gauss**

Links to an instance of the software package Gauss. This object performs the whole communication with Gauss and has total control over Gauss, i.e. it starts, commands, and ends Gauss.

Declaration

public class Gauss **extends** java.lang.Object

Fields

public static boolean isDesignTime

Flags design time. Set to false by default. If set to true this object will not try to load the native code and to start Gauss.

public static String compileCFG
Name of the configuration file for compiling Gauss sourcen.
public boolean debug

Constructors

public Gauss( )
Constructs Gauss object without starting Gauss.

Methods

public void addPropertyChangeListener
(`java.beans.PropertyChangeListener` **l**)

public synchronized void addPropertyChangeListener
(`java.lang.String` **propertyName**,
`java.beans.PropertyChangeListener` **listener**)

public boolean checkForMissingValues (`gauss.GaussData` **arg**)
Checks whether the argument contains any missig values.

public boolean executeCommand (`java.lang.String` **command**)
Preferred method to execute a Gauss command. This method blocks until Gauss finished the command execution and returns the success of the execution. A command is executed successfully if no error is reported to the Gauss errorlog file.

public void firePropertyChange (`java.lang.String` **propertyName**,
`java.lang.Object` **oldValue**,
`java.lang.Object` **newValue**)

public String getCurrentThreadName( )
Returns the name of the currently executing GaussThread or GaussProcedureCall or `null` if Gauss is idle (waits for jobs).

public String getDLibraryList (`java.lang.String` **dlibrary**)
Returns a list of the DLLs currently loaded by Gauss.

public String getGcgName( )
Returns the file name that contains the compiled source code.

public String getInstanceName( )
Returns the instance name of the associateted GAUSS program.

public String getInstanceTempDir( )
Returns the temporary directory used by the running Gauss instance. That directory is used to place the configuration, errorlog, and graphic files of the

running Gauss program.

public String getInstanceTempDirDoubleBackSlash( )
Returns the temporary directory used by the running Gauss instance with two backslashes instead of one. This is particular useful in Gauss program code.

public boolean getIsWorking( )
Gets the isWorking property (boolean) value.

public String getJmPath( )
Returns the root directory.

public String getJmPathDoubleBackSlash( )
Returns the root directory with two backslashes instead of one. This is particular useful in Gauss program code.

public String getLibraryList (`java.lang.String` **library**)
Returns a list of Gauss libraries loaded by Gauss with the *library* command. Returns `null` if not run in debug-mode.

public String getTempPath( )
Returns the temp root path.

public synchronized boolean hasListeners
(`java.lang.String` **propertyName**)

public native boolean isExecuting( )
Checks the status of the connected Gauss program.

public boolean loadDlibrary (`java.lang.String` **library**)
This method loads a new DLL to Gauss. The DLL is added to the already loaded DLLs. If run in designTime-mode nothing is done and the method returns false by default.

public void loadLibrary (`java.lang.String` **library**)
This method loads a Gauss library if run in debug-mode.

public GaussData readGaussData (`java.lang.String` **arg**)
Returns a symbol from Gauss. Only strings, string arrays, and matrices can be accessed. If the symbol does not exist in Gauss or is not of type matrix, string, or string array a message is displayed and null returned. This method blocks until all data is written from Gauss to Java, i.e. when this method returns non-null the return object can savely be accessed.

public GaussData readGaussDataArray (`java.lang.String[]` **arg**)
Returns one or more symbols from Gauss at the same time. Only strings, string arrays, and matrices can be accessed. If the symbol does not exist in

Gauss or is not of type matrix, string, or string array the corresponding element in the return array contains `null`. When reading *N>1* symbols from Gauss it is advantageous to use this method instead of calling *N* times `readGaussData()` in terms of performance. This method blocks until all data is written from Gauss to Java, i.e. when this method returns all non-null elements of the return object can savely be accessed.

public void register (`gauss.GaussProcedureCall` **gpc**)
Registers a GaussProcedureCall. If `isWorking()` returns true the argument is added to the thread list and gets started when all threads registered before finished. Otherwise the argument is executed immediately.

public void register (`gauss.GaussThread` **t**)
Registers a GaussThread. If `isWorking()` returns true the argument is added to the thread list and gets started when all threads registered before finished. Otherwise the argument is executed immediately.

public boolean remove (`java.lang.Thread` **t**)
Removes the specified thread either from Gauss (i.e. stops the argument if running) or form the list of unprocessed threads.

public void removePropertyChangeListener
(`java.beans.PropertyChangeListener` **l**)

public synchronized void removePropertyChangeListener
(`java.lang.String` **propertyName**,
`java.beans.PropertyChangeListener` **listener**)

public void setCurrentThreadName
(`java.lang.String` **newCurrentThreadName**)
Sets the name of the currently executing thread.

public void setGcgName (`java.lang.String` **newGcgName**)
Sets the name of the file that contains the compiled source code.

public void setOwnerComponent
(`java.awt.Component` **newOwnerComponent**)
Sets the component that initiated a Gauss job.

public void startGauss( )
Method to start Gauss.
This is the preferred method to start Gauss. A new instance of the Gauss program is started and attach to this object.

public void stopExecution( )
Preferred method to stop current program execution. The next Gauss job from the queue (if there is one) is started.

public boolean terminateGauss( )
Preferred termination method. Gauss is finished without confirmation if there are no running gauss programs and no waiting gauss jobs. Otherwise a confirmation dialog pops up.

public boolean writeGaussData (`gauss.GaussData` **arg**)
Preferred method to write a GaussData object to Gauss. When this method succeeds the data of the argument object is assigned to a global symbol in Gauss with the name `arg.getSymbolName()`
This method blocks until all data is written to Gauss, i.e. when this method returns true the next operation in Gauss can rely on a global symbol named `arg.getSymbolName()`
The GaussData argument can be uninitialized (in Gauss: y={}).

public boolean writeGaussDataArray (`gauss.GaussData[]` **arg**)
Preferred method to write more than one GaussData objects to Gauss. When this method succeeds the data of the argument objects are assigned to global symbols in Gauss with the name `arg[i].getSymbolName()`
This method blocks until all data is written to Gauss, i.e. when this method returns true the next operation in Gauss can rely on the global symbols named `arg[i].getSymbolName()`
The GaussData arguments can be uninitialized (in Gauss: y={}).

Class **GaussData**

Representation of Gauss software data symbols (matrix, string, and string array). It also can be used as a table model.
At construction time the name of the GaussData object is specified. It cannot be changed later. In the following the term *data* is used for the numbers or strings that can be stored in this object.
On some UI-component `GaussData` objects can be conveniently displayed with

- `GaussDataTable`,
- `GaussDataTextField`, and
- `GaussDataLabel` objects.

All centrally accessable `GaussData` objects are administered by a `GaussSymbolTable` object.

Declaration

public class GaussData **extends** javax.swing.table.AbstractTableModel
implements java.lang.Cloneable

Fields

public static final String MATRIX_CHANGE
public static final String STRING_CHANGE
public static final String STRUCTURE_CHANGE
public static final int NOT_INITIALIZED
Return value of `getType()` if this object contains no data.
public static final int MATRIX
Return value of `getType()` if this object contains numeric data.
public static final int STRING
Return value of `getType()` if this object contains a string.
public static final int STRING_ARRAY
Return value of `getType()` if this object contains a two dimensional array of strings.
public static final int MISSING_VALUE
Input value for constructors in order to construct missing values GaussData. The value stored is a NaN (Not a Number).
public static final int ONES
Input value for constructors in order to construct a matrix of ones.
public static final int ZEROS
Input value for constructors in order to construct a matrix of zeros.
public static final int EMPTY
Identifier for empty GaussData objects.

Constructors

public GaussData (`double[][]` **data**,
`java.lang.String` **name**)
Constructs a 2-dimensional matrix if the data argument is not empty or null.

public GaussData (`double[]` **data**,
`java.lang.String` **symbolName**)
Constructs a row vector if the data argument is not empty or null.

public GaussData (`double` **data**,
`java.lang.String` **symbolName**)
Constructs a scalar.

public GaussData (`int[][]` **data**,
`java.lang.String` **name**)
Constructs a 2-dimensional matrix if the data argument is not empty or null. The integers are internally converted to doubles.

public GaussData (`java.lang.String` **symbolName**)
Constructs a uninitialized GaussData object.

public GaussData (`java.lang.String[][]` **data**,
`java.lang.String` **name**)
Constructs a 2-dimensional array of strings if the data argument is not empty or null.

public GaussData (`java.lang.String[]` **value**,
`java.lang.String` **name**)
Constructs a row vector of strings if the data argument is not empty or null.

public GaussData (`java.lang.String` **value**,
`java.lang.String` **symbolName**)
Constructs a string object if the data argument is not null.

Methods

public synchronized void addPropertyChangeListener
(`java.beans.PropertyChangeListener` **l**)

public synchronized void addPropertyChangeListener
(`java.lang.String` **propertyName**,
`java.beans.PropertyChangeListener` **l**)

public synchronized void addVetoableChangeListener
(`java.beans.VetoableChangeListener` **l**)

public void appendColumn (`double` **d**)
Appends a column to the end of the data. In Gauss one would code `x=x y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void appendColumn (`double[]` **xx**)
Appends a column to the end of the data. In Gauss one would code `x=x y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void appendColumn (`java.lang.String` **string**)
Appends a column to the end of the data. In Gauss one would code `x=x$ y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with

`SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void appendColumns (`double[][] x`)
Appends a column to the end of the data. In Gauss one would code `x=x y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void appendColumns (`gauss.GaussData` **gaussData**)
Appends a column to the end of the data. In Gauss one would code `x=x y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void appendColumns (`java.lang.String[][] x`)
Appends a column to the end of the data. In Gauss one would code `x=x$ y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void appendRow (`double` **d**)
Appends a row to the end of the data. In Gauss one would code `x=x|y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is redirected to it. In this case returning from this method is no guarantee that the operation is finished.

public void appendRow (`java.lang.String` **newElement**)
Appends a row to the end of the data. In Gauss one would code `x=x$|y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is redirected to it. In this case returning from this method is no guarantee that the operation is finished.

public void appendRows (`double[][]` **rows**)
Appends rows to the end of the data. In Gauss one would code `x=x|y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is redirected to it. In this case returning

from this method is no guarantee that the operation is finished.

public void appendRows (`gauss.GaussData` **gaussData**)
Appends rows to the end of the data. In Gauss one would code `x=x|y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is redirected to it. In this case returning from this method is no guarantee that the operation is finished.

public void appendRows (`java.lang.String[][]` **rows**)
Appends rows to the end of the data. In Gauss one would code `x=x$|y;` To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is redirected to it. In this case returning from this method is no guarantee that the operation is finished.

public Object clone( )
Returns a new GaussData object with a the same symbol name and the same data but leaves behind the listeners.

public GaussData clone (`java.lang.String` **newName**)
Returns a new GaussData object with a new symbol name. It contains the same data as in this object but leaves behind the listeners.

public void fireTableChanged (`javax.swing.event.TableModelEvent` **e**)
Forward the given notification event to all TableModelListeners that registered themselves as listeners for this table model.

public int getCols( )
Returns the number of columns.

public Object getColumnAt (`int` **c**)
Returns a column at the specified index. The type of the return object is either `double[]` or `String[]`

public int getColumnCount( )
Returns the number of columns.

public double getDoubleValueAt (`int` **row**,
`int` **column**)
Returns the double-element at index `[row,column]`.

public int getIndexOfElement (`double` **d**)
Returns the index of the first ocurrence of the argument in GaussData. The search is conducted row-wise, i.e. the first row is searched, then the second, etc. If the search algorithm finds a matrix element such that m[i][j]==d, the indices i and j are returned as an array of integers with first

the row index (i) and then the column index (j). Note that indices start at 0 and end at getRows()-1 / getCols()-1. If the search fails or the GaussData object represents a string array, int[] {-1,-1} is returned.
This method should be called from the event dispatching thread.

public int getIndexOfElement (`double` **d**,
`int` **startRowIndex**,
`int` **startColumnIndex**)

Returns the index of the first occurrence of the argument in this GaussData object, starting the search at the specified row and column index.
The search is conducted row-wise, i.e. the row at startRowIndex is searched first, then the row at startRowIndex + 1, etc. If the search algorithm finds a matrix element such that m[i][j]==d, the indices i and j are returned as an array of integers with first the row index (i) and then the column index (j). Note that indices start at 0 and end at getRows()-1 / getCols()-1. If the search fails or the GaussData object represents a string array, int[] {-1,-1} is returned.
This method should be called from the event dispatching thread. See javax.swing.SwingUtilities.invokeLater().

public int getIndexOfElement (`java.lang.String` **string**)

Returns the index of the first occurrence of the argument in GaussData.
The search is conducted row-wise, i.e. the first row is searched, then the second, etc. If the search algorithm finds a matrix element such that m[i][j]==string, the indices i and j are returned as an array of integers with first the row index (i) and then the column index (j). Note that indices start at 0 and end at getRows()-1 / getCols()-1. If the search fails or the GaussData object represents a matrix, int[] {-1,-1} is returned.
This method should be called from the event dispatching thread. See javax.swing.SwingUtilities.invokeLater().

public int getIndexOfElement (`java.lang.String` **arg**,
`int` **startRowIndex**,
`int` **startColumnIndex**)

Returns the index of the first ocurrence of the argument in this GaussData object, starting the search at the specified row and column index.
The search is conducted row-wise, i.e. the row at startRowIndex is searched first, then the row at startRowIndex + 1, etc. If the search algorithm finds a matrix element such that m[i][j]==d, the indices i and j are returned as an array of integers with first the row index (i) and then the column index (j). Note that indices start at 0 and end at getRows()-1 / getCols()-1. If the search fails or the GaussData object represents a matrix, int[] {-1,-1} is returned.
This method should be called from the event dispatching thread. See javax.swing.SwingUtilities.invokeLater().

public int getIntValueAt (`int` **row**,
`int` **column**)
Returns the double-element at index `[row,column]` truncated to an integer.

public String getName( )
Returns the symbol name of this GaussData object. The symbol name is immutable.

public int getRowCount( )
Returns number of rows.

public int getRows( )
Returns number of rows.

public String getStringValueAt (`int` **row**,
`int` **column**)
Returns the String-element at index `[row,column]`.

public TableModelListener getTableModelListener( )
Returns all registered TableModelListeners. The returned array has length 0 if no listeners are registered.

public int getType( )
Returns the data type of a GaussData object.
The result equals the GAUSS command `type(X);`

public Object getValueAt (`int` **row**,
`int` **column**)
Returns an attribute value for the cell at index `[row,column]`.
row must be in the range 0,...,*getRows()-1*, *column* in the range 0,...,*getCols()-1*.

Note that the primitive type double is wrapped by java.lang.Double. If `isMatrix()` returns true, it is more efficient to call `getDoubleValueAt()` or `getIntValueAt()`

public Object getValues( )
Returns data Object of this GaussData-Object. The return value has format double[][] in case of MATRIX and String[][] in case of STRING and STRING_ARRAY, or `null` in case of NOT_INITIALIZED.

public boolean isCellEditable (`int` **r**,
`int` **c**)
Returns `isEditable()`.

public boolean isEditable( )
Returns whether the data can be changed or not.

public boolean isMatrix( )
Indicates whether GaussData is of type `MATRIX`.

public boolean isString( )
Indicates whether GaussData is of type `STRING` or `STRING_ARRAY`

public boolean isThreadSafe( )
Short for `SwingUtilities.isEventDispatchThread()`

public static boolean isValidName (`java.lang.String` **name**)
Helper method to see whether the string argument is a valid symbol name in Gauss. This method does not check whether the argument string is a Gauss command or procedure or function name.

public static double makeDoubleMatrix (`int` **rows**,
`int` **cols**,
`int` **type**)
Creates a 2-dimensional array of missing values, zeros, or ones.

public void print( )
Prints this GaussData object to `System.out`

public void print (`java.io.PrintStream` **p**)
Prints this GaussData object to the argument print stream.

public void print (`java.io.PrintStream` **p**,
`int` **width**,
`int` **precision**,
`boolean` **transposed**)
Prints this GaussData object to the argument print stream, with minimum field width, and precision. Output can be made transposed.

public void removeColumns (`int` **index0**,
`int` **index1**)
Removes all columns from `index0` to `index1`.

public void removePropertyChangeListener
(`java.beans.PropertyChangeListener` **l**)

public void removePropertyChangeListener
(`java.lang.String` **propertyName**,
`java.beans.PropertyChangeListener` **l**)

public void removeRows (`int` **index0**,
`int` **index1**)
Removes all rows from `index0` to `index1`.

public void removeVetoableChangeListener
(`java.beans.VetoableChangeListener` **l**)

public void setDoubleValueAt (`double` **newValue**,
`int` **row**,
`int` **col**)

Changes the element at index `[row,col]` to newValue. To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void setIntValueAt (`int` **intValue**,
`int` **row**,
`int` **col**)

Changes the element at index `[row,col]` to intValue. To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void setStringValueAt (`java.lang.String` **value**,
`int` **row**,
`int` **col**)

Changes the element at index `[row,col]` to the specified string value. To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void setValueAt (`java.lang.Object` **value**,
`int` **row**,
`int` **col**)

Changes the element at index `[row,col]` to the value object. To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void setValues (`double[][]` **newValues**)

Changes data to newValues. To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void setValues (`gauss.GaussData` **values**)

Changes data to GaussData values. To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void setValues (`java.lang.String[][]` **newValues**)
Changes data to new string values. To make sure that the data manipulation is done when this method returns, call this method from the event dispatching thread. If this method is not called from the event dispatching thread it is called again with `SwingUtilities.invokeLater()`. In this case returning from this method is no guarantee that the operation is finished.

public void syncDimension (`int` **newRows**,
`int` **newCols**,
`double` **fillValueDouble**,
`java.lang.String` **fillValueString**)
Fits data field to new dimension.

public String toString( )
Returns an informative string representation for this GaussData object.

Class **GaussDataLabel**

Displays elements of a GaussData object.
Objects of this class can display up to two elements of GaussData object(s). The elements are referred to as *firstArg* and *secondArg*. They are displayed using a C-style `sprinf()` *formatString*.

Declaration

public class GaussDataLabel **extends** javax.swing.JLabel
implements javax.swing.event.TableModelListener

Constructors

public GaussDataLabel( )
GaussDataLabel constructor.

Methods

public int getFirstArgCol( )

public int getFirstArgRow( )

public String getFirstArgSymbolName( )

public String getFormatString( )

public int getSecondArgCol( )

public int getSecondArgRow( )

public String getSecondArgSymbolName( )

public boolean isLocalGaussData( )

public void paint (`java.awt.Graphics` **g**)

public void setFirstArgCol (`int` **newFirstArgCol**)

public void setFirstArgRow (`int` **newFirstArgRow**)

public void setFirstArgSymbolName
(`java.lang.String` **newFirstArgSymbolName**)

public void setFormatString (`java.lang.String` **newFormatString**)

public void setLocalGaussData (`boolean` **newLocalGaussData**)

public void setSecondArgCol (`int` **newSecondArgCol**)

public void setSecondArgRow (`int` **newSecondArgRow**)

public void setSecondArgSymbolName
(`java.lang.String` **newSecondArgSymbolName**)

public void tableChanged (`javax.swing.event.TableModelEvent` **e**)

Class **GaussDataOperators**

Collection of often used operators.
The operators can be applied on GaussData objects.

Declaration

public class GaussDataOperators **extends** java.lang.Object

Constructors

public GaussDataOperators( )
GaussDataOperators constructor.

Methods

public static void concatenateArg (`gauss.GaussData` **arg**,
`java.lang.String` **appendix**)

public static void concatenateArg (`java.lang.String` **prefix**,
`gauss.GaussData` **arg**)

public static GaussData deleteRow (`gauss.GaussData` **arg1**,
`int` **arg2**)

public static void deleteRowArg (`gauss.GaussData` **arg1**,
`int` **arg2**)

public static GaussData deleteRowsIf (`gauss.GaussData` **arg1**,
`double[][]` **arg2**)

public static GaussData indexSelection (`gauss.GaussData` **arg**,
`gauss.GaussData` **rowIdx**,
`gauss.GaussData` **colIdx**)

public static GaussData setDifference (`gauss.GaussData` **arg1**,
`gauss.GaussData` **arg2**)

Class **GaussDataTable**

Displays the contents of a `GaussData` object.
It is recommended to use `GaussDataView` objects for displaying Gauss data.
Support classes are

- GaussDataTableDoubleEditor
- GaussDataTableStringEditor
- GaussDataTableDoubleRenderer
- GaussDataTableStringRenderer

Declaration

public class GaussDataTable **extends** util.component.BorderTable
implements java.awt.event.ActionListener

Fields

public static final String ENTER
ENTER - command
public static final String CANCEL
CANCEL - command

public boolean columnHeader
public int rowSelectionModus

Constructors

public GaussDataTable( )

public GaussDataTable (`javax.swing.table.TableModel` **dm**)

public GaussDataTable (`javax.swing.table.TableModel` **dm**,
`boolean` **link**)

public GaussDataTable (`javax.swing.table.TableModel` **dm**,
`javax.swing.table.TableColumnModel` **cm**)

public GaussDataTable (`javax.swing.table.TableModel` **dm**,
`javax.swing.table.TableColumnModel` **cm**,
`boolean` **link**)

Methods

public void actionPerformed (`java.awt.event.ActionEvent` **e**)

public void addNotify( )

public void addTableModelListener
(`javax.swing.event.TableModelListener` **l**)

This method adds a model listener to the current GaussData object. When the model changes (setModel()) the model listeners registered with this method are removed from the old GaussData object and added to the new GaussData object. The model listeners stick to the GaussDataTable rather then to the GaussData.

public void createDefaultColumnsFromModel( )

This method will create default columns for the table from the data model using the getColumnCount() and getColumnClass() methods defined in the TableModel interface.
This method will clear any exsiting columns before creating the new columns based on information from the model.

public JTableHeader createDefaultTableHeader( )

Returns the default table header object which is a JTableHeader. Subclass can override this method to return a different table header object

public boolean editCellAt (`int` **row**,
`int` **column**,
`java.util.EventObject` **e**)

Programmatically starts editing the cell at *row* and *column*, if the cell is editable. To prevent the JTable from editing a particular table, column or cell value, return false from the isCellEditable() method in the TableModel interface.

public TableCellRenderer getCellRenderer( )

public TableCellEditor getDefaultEditor (`java.lang.Class` **columnClass**)
Returns the editor to be used when no editor has been set in a TableColumn. During the editing of cells the editor is fetched from a Hashtable of entries according to the class of the cells in the column. If there is no entry for this *columnClass* the method returns the entry for the most specific superclass. The JTable installs entries for *Object*, *Number* and *Boolean* all which can be modified or replaced.

public TableCellRenderer getDefaultRenderer
(`java.lang.Class` **columnClass**)
New implementation because field 'defaultRenderersByColumnClass' is not initialized in this subclass and this method is used by the maxCellWidth methods

public double getDoubleValueAt (`int` **row**,
`int` **column**)

public boolean getEditable( )
Returns the property editable.

public TableCellEditor getEditor( )

public JScrollPane getEnclosingScrollPane( )

public Gauss getGauss( )
Implementation of interface function from interface GaussObject.
Returns null by default.

public GaussData getGaussDataSelectedRows (`java.lang.String` **name**)

public GaussSymbolTable getGaussSymbolTable( )
Implementation of interface function from interface GaussObject.
Returns current `GaussSymbolTable`.

public int getPrecision( )
If table displays numeric data, this method returns number of digits right of decimal point.

public int getRowSelectionModus( )

public int getScrollableBlockIncrement (`java.awt.Rectangle` **visibleRect**,
`int` **orientation**,
`int` **direction**)

Returns The visibleRect.height or visibleRect.width, depending on the table's orientation.

public final boolean getScrollableTracksViewportHeight( )

Returns false to indicate that the height of the viewport does not determine the height of the table.
This implementation is assumed by GaussDataView and its implementing LayoutManager GDVScrollPaneLayout.

public final boolean getScrollableTracksViewportWidth( )

Returns false to indicate that the width of the viewport does not determine the width of the table.
This implementation is assumed by GaussDataView and its implementing LayoutManager GDVScrollPaneLayout.

public int getScrollableUnitIncrement (`java.awt.Rectangle` **visibleRect**,
`int` **orientation**,
`int` **direction**)

Returns the scroll increment that completely exposes one new row or column (depending on the orientation).
This method is called each time the user requests a unit scroll.
End points of scrollbar clicked.

public String getStringValueAt (`int` **row**,
`int` **column**)

public String getSymbolName_indexSelectedRows( )

public String getSymbolName_lowerBound( )

public String getSymbolName_selectedRows( )

public String getSymbolName_upperBound( )

public String getSymbolName( )

Returns the symbol name of the displayed data.

public int getType( )

Returns the type of the displayed data.

public Verifier getVerifier( )

Returns the verifier object of this table.

public boolean isCellEditable (`int` **row**,
`int` **col**)

New implementation of JTable.isCellEditable() method.

public boolean isEditable( )

public boolean isLocalGaussData( )

public Component prepareRenderer
(`javax.swing.table.TableCellRenderer` **renderer**,
`int` **row**,
`int` **column**)

public void setAutoResizeMode (`int` **mode**)

public void setCellRenderer
(`javax.swing.table.TableCellRenderer` **newCellRenderer**)

public void setColumnHeader (`boolean` **newColumnHeader**)

public void setEditable (`boolean` **editable**)
Enable or disable editable property of this GaussDataTable

public void setEditor (`javax.swing.table.TableCellEditor` **arg**)

public void setGauss (`gauss.Gauss` **g**)
Implementation of interface method from interface GaussObject.
Empty method body.

public void setGaussSymbolTable (`gauss.GaussSymbolTable` **t**)
Implementation of interface method from interface GaussObject.
Sets new GaussSymbolTable.

public void setInitialValues (`double[][]` **values**)

public void setInitialValues (`java.lang.String[][]` **values**)

public void setLocalGaussData (`boolean` **newLocalGaussData**)

public void setModel (`javax.swing.table.TableModel` **newTableModel**)
Reimplements JTable.setModel() method.
`JTable` already implemented `TableModelListener` and registeres/ unregisters itself to new/ old `TableModels`. All desired operations at a model change must therefore coded in the method *tableChanged()*.
This method checks for a `null` TableModel argument and returns immediately if this is present.

public void setPrecision (`int` **precision**)
If table displays numeric data, this method sets number of digits right of decimal point.

public void setRowSelectionModus (`int` **newRowSelectionModus**)

public void setSymbolName_indexSelectedRows (`java.lang.String` **arg**)

public void setSymbolName_lowerBound
(`java.lang.String` **newSymbolName_lowerBound**)

public void setSymbolName_selectedRows
(`java.lang.String` **newSymbolName_selectedRows**)

public void setSymbolName_upperBound
(`java.lang.String` **newSymbolName_upperBound**)

public void setSymbolName (`java.lang.String` **name**)
Sets symbol name of new GaussData object that is displayed by this table.

public void setSymbolName
(`java.lang.String` **name**,
`gauss.GaussSymbolTable` **localSymbolTable**)

public void setSymbolNameForColumnHeader (`java.lang.String` **name**)

public void setVerifier (`util.component.Verifier` **newVerifier**)
Sets a new verifier object for this table.

public void tableChanged (`javax.swing.event.TableModelEvent` **e**)

public void updateUI( )

public void valueChanged (`javax.swing.event.ListSelectionEvent` **e**)
Invoked when row selection changes.
The data object selected rows (symbolName_selectedRows) is updated.

Class **GaussDataTableCellRenderer**

Support class for GaussDataTable.

Renderer component for gauss.GaussDataTable when cell value is java.lang.Double

Declaration

public class GaussDataTableCellRenderer **extends** javax.swing.JLabel
implements javax.swing.table.TableCellRenderer

Constructors

public GaussDataTableCellRenderer( )

Methods

public int getPrecision( )

public Component getTableCellRendererComponent
(`javax.swing.JTable` **table**,
`java.lang.Object` **value**,
`boolean` **isSelected**,
`boolean` **hasFocus**,
`int` **row**,
`int` **column**)

public Color getUnselectedBackground( )

public Color getUnselectedForeground( )

public void setPrecision (`int` **precision**)

public void setUnselectedBackground
(`java.awt.Color` **newUnselectedBackground**)

public void setUnselectedForeground
(`java.awt.Color` **newUnselectedForeground**)

Class **GaussDataTableScrollPane**

Use this class to view a potentially large `GaussDataTable` in a potentially small place.
Potentially because the view can encompass the entire `GaussDataTable` or just a piece of it. In the latter case one can scroll through the whole `GaussDataTable`.
A GaussDataView that becomes scrollbars if more data than displayed columns/ rows become available.
This object is preferred to the simple `GaussDataView`!! The FixedScrollPane is an extended JScrollPane that allows to set column header and row header policies. For laying out the scroll pane components it uses either the FixedScrollPaneLayout or the TightScrollPaneLayout. Both layout managers preserve space for scrollbars and row/ column headers if there is a theoretical chance for them to appear (i.e their policy variable is not set to some kind of NEVER).
minimumVisibleRows = no less than 'minimumVisibleRows' are displayed if there is not enough data to display, space is reserved. maximumVisibleRows = no more than 'maximumVisibleRows' are displayed if there is not enough space, display as much rows as possible.

Declaration

public class GaussDataTableScrollPane **extends** javax.swing.JScrollPane

Constructors

public GaussDataTableScrollPane( )

public GaussDataTableScrollPane (`gauss.GaussDataTable` **gdt**)

Methods

public int getColumnHeaderAlignment( )

public TableCellRenderer getColumnHeaderCellRenderer( )

public int getColumnHeaderDataFromModel( )

public GaussData getColumnHeaderGaussData( )

public GaussDataTable getColumnHeaderGaussDataTable( )

public int getColumnHeaderPolicy( )

public TableCellRenderer getColumnHeaderRenderer( )

public String getColumnHeaderStringData( )

public String getColumnHeaderSymbolName( )

public int getMaximumVisibleColumns( )

public int getMaximumVisibleRows( )

public int getMinimumVisibleColumns( )

public int getMinimumVisibleRows( )

public int getRowHeaderAlignment( )

public TableCellRenderer getRowHeaderCellRenderer( )

public int getRowHeaderDataFromModel( )

public GaussData getRowHeaderGaussData( )

public int getRowHeaderPolicy( )
Returns row header policy.

public String getRowHeaderSymbolName( )

public boolean isColumnHeaderDefaultData( )

public boolean isLocalGaussData( )

public boolean isRowHeaderDefaultData( )

public void setBorder (`javax.swing.border.Border` **border**)

public void setColumnHeaderAlignment
(`int` **newColumnHeaderAlignment**)

public void setColumnHeaderCellRenderer
(`javax.swing.table.TableCellRenderer` **newColumnHeaderCellRenderer**)

public void setColumnHeaderDataFromModel
(`int` **newColumnHeaderDataFromModel**)

public void setColumnHeaderDefaultData
(`boolean` **newColumnHeaderDefaultData**)

public void setColumnHeaderGaussData
(`gauss.GaussData` **newColumnHeaderGaussData**)

public void setColumnHeaderPolicy (`int` **policy**)
Determines when the column header appears in the scrollpane. The options are:

- JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
- JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
- JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS

public void setColumnHeaderStringData
(`java.lang.String[]` **newColumnHeaderStringData**)

public void setColumnHeaderSymbolName
(`java.lang.String` **newColumnHeaderSymbolName**)

public void setLocalGaussData (`boolean` **newLocalGaussData**)

public void setMaximumVisibleColumns
(`int` **newMaximumVisibleColumns**)

public void setMaximumVisibleRows (`int` **newMaximumVisibleRows**)

public void setMinimumVisibleColumns
(`int` **newMinimumVisibleColumns**)

public void setMinimumVisibleRows (`int` **newMinimumVisibleRows**)

public void setRowHeaderAlignment (`int` **newRowHeaderAlignment**)

public void setRowHeaderCellRenderer
(`javax.swing.table.TableCellRenderer` **newRowHeaderCellRenderer**)

public void setRowHeaderDataFromModel
(`int` **newRowHeaderDataFromModel**)

public void setRowHeaderDefaultData
(`boolean` **newRowHeaderDefaultData**)

public void setRowHeaderGaussData
(`gauss.GaussData` **newRowHeaderGaussData**)

public void setRowHeaderPolicy (`int` **policy**)

public void setRowHeaderSymbolName
(`java.lang.String` **newRowHeaderSymbolName**)

public void setRowHeaderWidth (`int` **width**)
Sets a new row header width.

public void setViewportView (`java.awt.Component` **view**)

Class **GaussDataTableScrollPaneLayout**

Layout manager for GaussDataTableScrollPane.

Declaration

public class GaussDataTableScrollPaneLayout
extends javax.swing.ScrollPaneLayout

Constructors

public GaussDataTableScrollPaneLayout( )

Methods

public void layoutContainer (`java.awt.Container` **parent**)

public Dimension minimumLayoutSize (`java.awt.Container` **parent**)

The minimum size of a ScrollPane is the size of the insets plus minimum size of the viewport, plus the scrollpane's viewportBorder insets, plus the minimum size of the visible headers, plus the minimum size of the scrollbars whose displayPolicy isn't NEVER.

public Dimension preferredLayoutSize (`java.awt.Container` **parent**)
The preferred size of a ScrollPane is the size of the insets, plus the preferred size of the viewport, plus the preferred size of the headers, plus the preferred size of the scrollbars that will appear given the current view and the current scrollbar displayPolicies.

public void syncWithScrollPane (`javax.swing.JScrollPane` **sp**)

Class **GaussDataTextField**

Editor component for GaussData objects.
This is basically a JTextField that is connected to a GaussData object.

Declaration

public class GaussDataTextField **extends** util.component.ModalTextField
implements javax.swing.event.TableModelListener

Constructors

public GaussDataTextField( )

public GaussDataTextField (`int` **columns**)

Methods

public void addNotify( )

public int getCol( )
Returns the column number of the referenced GaussData object. Default is 1.

public double getInitialNumber( )
Returns the initial number.

public String getInitialString( )
Returns the initial string.

public Dimension getMaximumSize( )

If `fixedSize() == true` return the preferredSize, otherwise return the maximumSize.

public Dimension getMinimumSize( )
If `fixedSize() == true` return the preferredSize, otherwise return the minimumSize.

public int getRow( )
Returns the row number of the referenced GaussData object. Default is 1.

public String getSymbolName_lowerBound( )
Returns the symbol name of the referenced lower bound GaussData object.

public String getSymbolName_upperBound( )
Returns the symbol name of the referenced upper bound GaussData object.

public String getSymbolName( )
Returns the symbol name of the referenced GaussData object.

public boolean isFixedSize( )
Return the fixed size property. If this method returns true, `getMinimumSize()` and `getMaximumSize()` return the components preferred size.

public boolean isLocalGaussData( )
Return the local data property.

public void message( )
Pops up a message dialog. Called by the validation mechanism if the event queue processes an event that shifts input focus permanently away from this component and the input text did not pass the verifier.

public void paint (`java.awt.Graphics` **g**)

public void setCol (`int` **newCol**)
Set the column number of the referenced GaussData object.

public void setFixedSize (`boolean` **newFixedSize**)
Set the fixed size property. If newFixedSize is true, `getMinimumSize()` and `getMaximumSize()` return the components preferred size. This behaviour can be advantegous for efficient layout management.

public void setInitialNumber (`double` **newInitialNumber**)
Set a new initial number. This number will be stored in the GaussSymbolTable when this component is added to the component hierarchy.

public void setInitialString (`java.lang.String` **newInitialString**)
Set a new initial string. This string will be stored in the GaussSymbolTable

when this component is added to the component hierarchy.

public void setLocalGaussData (`boolean` **newLocalGaussData**)
Set the local data property. If set to true, the component tries to get the GaussDataObject from a local GaussDataTable.

public void setRow (`int` **newRow**)
Set the row number of the referenced GaussData object.

public void setSymbolName_lowerBound
(`java.lang.String` **newSymbolName_lowerBound**)
Sets the symbol name of the referenced lower bound GaussData object.

public void setSymbolName_upperBound
(`java.lang.String` **newSymbolName_upperBound**)
Sets the symbol name of the referenced upper bound GaussData object.

public void setSymbolName (`java.lang.String` **newSymbolName**)
Sets the symbol name of the referenced GaussData object.

public void tableChanged (`javax.swing.event.TableModelEvent` **e**)
Calls `updateText()`

public boolean validateInput( )
Checks whether lower and upper bound GaussData object(s) are referenced and then calls super method.

Class **GaussFrame**

The top level GaussGUI component in the Gauss GUI hierarchy.
It is characterized with two properties `Gauss`, and `GaussSymbolTable`. These properties can be accessed by all lower level GaussGUI components in the hierarchy with `getXxxx()` methods.

When creating a `GaussFrame` object the constructor tries to start an instance of the Gauss program that runs in the background as a number cruncher slave.

Declaration

public class GaussFrame **extends** javax.swing.JFrame
implements GaussEngineRoot,
GaussSymbolRoot

Constructors

public GaussFrame( )

public GaussFrame (`java.lang.String` **title**)

Methods

public void addNotify( )

public void closeGaussFrame( )

public Gauss getGauss( )
Returns reference to object that keeps communication with the Gauss program.

public GaussSymbolTable getGaussSymbolTable( )
Returns symbol table of this gui layer.

public boolean isConfirmClosingOperation( )
Returns true if closing operation of this frame must be confirmed by the user.

public boolean isSystemExitWhenClosing( )
Returns true, if closing operation means the termination of the complete Java application.

public void setConfirmClosingOperation
(`boolean` **newConfirmClosingOperation**)
Specify whether closing request of this frame must be confirmed by the user.

public void setSystemExitWhenClosing
(`boolean` **newSystemExitWhenClosing**)
Specify whether closing request of this frame equals to complete program termination.

Class **GaussInternalFrame**

JInternalFrame with a GaussSymbolTable object.
This class has the special property that it holds an own `GaussSymbolTable` object that can be accessed by its childs. This is necessary for not mixing up data of different internal frame instances that access the `GaussSymbolTable` of the `GaussFrame`. `setDefaultCloseOperation()` with arguments DO_NOTHING_ON_CLOSE or DISPOSE_ON_CLOSE will bring up a confirm dialog that asks for disposing confirmation when the internal frame is closed by the user. The HIDE_ON_CLOSE option closes and removes the internal frame without asking for permission (default setting). Note that in all cases the internal frame is removedfrom the desktop pane. For repeatedly showing this frame a external reference on this object must be kept. See remark.

Remark:In Java 1.2 JInternalFrame has bug 4138031. This class provides a workaround for it. (It seems to be fixed in Java 1.3)

Declaration

public class GaussInternalFrame **extends** javax.swing.JInternalFrame
implements GaussSymbolRoot

Constructors

public GaussInternalFrame( )

public GaussInternalFrame (`java.lang.String` **title**)

public GaussInternalFrame (`java.lang.String` **title**,
`boolean` **resizable**)

public GaussInternalFrame (`java.lang.String` **title**,
`boolean` **resizable**,
`boolean` **closable**)

public GaussInternalFrame (`java.lang.String` **title**,
`boolean` **resizable**,
`boolean` **closable**,
`boolean` **maximizable**)

public GaussInternalFrame (`java.lang.String` **title**,
`boolean` **resizable**,
`boolean` **closable**,
`boolean` **maximizable**,
`boolean` **iconifiable**)

Methods

public void addNotify( )
Links this instance to the tree of GaussSymbolTables.

public GaussSymbolTable getGaussSymbolTable( )
Returns symbol table of this gui layer.

Class **GaussObjectLinker**

Provides static methods for finding contextual correct GaussSymbolTable- and Gauss- objects at run time.

Declaration

public class GaussObjectLinker **extends** java.lang.Object

Methods

public static Gauss findGauss (`java.awt.Component` **c**)

Returns Gauss object at run time by walking up the component hierarcy.

public static GaussSymbolTable findGaussSymbolTable (`java.awt.Component` **c**)

Returns contextual right GaussSymbolTable object at run time by walking up the component hierarcy.

public static GaussSymbolTable findGaussSymbolTable (`java.awt.Component` **c**, `boolean` **local**)

Returns contextual right local GaussSymbolTable object at run time by walking up the component hierarcy.

Class **GaussPanel**

This class is the superclass for all Gauss user interfaces. It can be seen as a convenient container class that allows to implement computation in Gauss easily.

Declaration

public class GaussPanel **extends** util.component.CardChangePanel
implements util.ProcedureCallListener,
util.ProcedureCallConstants,
GaussThreadListener,
java.beans.PropertyChangeListener,
java.beans.VetoableChangeListener

Constructors

public GaussPanel( )

public GaussPanel (`boolean` **isDoubleBuffered**)

public GaussPanel (`java.awt.LayoutManager` **layout**)

public GaussPanel (`java.awt.LayoutManager` **layout**,
`boolean` **isDoubleBuffered**)

METHODS

public void addNotify( )
Notification to this component that it now has a parent component. When this method is invoked, the chain of parent components is set up with Keyboard-Action event listeners.

public void fireVetoableChange (`java.lang.String` **name**,
`boolean` **oldValue**,
`boolean` **newValue**)

public void gaussExec (`java.lang.String` **command**)
NOT THREAD SAVE!!

Execute Gauss command line from this GaussPanel.

public void gaussLoadLibrary (`java.lang.String` **library**)
Execute Gauss command line from this GaussPanel.

public void gaussRead (`java.lang.String` **symbolName**)
NOT THREAD SAVE!!

Read variable from Gauss to this GaussPanel.

public void gaussReadArray (`java.lang.String[]` **symbolNames**)
NOT THREAD SAVE!!

Read variable from Gauss to this GaussPanel.

public void gaussReadArray (`java.util.Vector` **vec**)
NOT THREAD SAVE!!

Read variable from Gauss to this GaussPanel.

public void gaussReadArrayLocal (`java.lang.String[]` **symbolNames**,
`gauss.GaussSymbolTable` **symbolTable**)
Read variable from Gauss to this GaussPanel.

public void gaussReadArrayLocal (`java.util.Vector` **vec**,
`gauss.GaussSymbolTable` **symbolTable**)
Read variable from Gauss to this GaussPanel.

public void gaussReadLocal (`java.lang.String` **symbolName**,
`gauss.GaussSymbolTable` **symbolTable**)
Read variable from Gauss to this GaussPanel.

public void gaussRun( )
NOT THREAD SAVE!!

Final method. Registers all previous gauss*-methods in a GaussThread.

public void gaussRun (`int` **actionType**)
NOT THREAD SAVE!!

Final method. Registers all previous gauss*-methods in a GaussThread.

public void gaussRun (`int` **actionType**,
`java.lang.Object[]` **param**)
NOT THREAD SAVE!!

Final method. Registers all previous gauss*-methods in a GaussThread.

public void gaussShow (`java.lang.String` **symbolName**,
`gauss.GaussDataTable` **target**)
Shows symbolName on target GaussDataView.

public void gaussStart (`java.lang.String` **stackName**)
NOT THREAD SAVE!!

Starts new Gauss thread.

public void gaussStart (`java.lang.String` **stackName**,
`boolean` **allowForMissingValues**)
NOT THREAD SAVE!!

Starts new Gauss thread.

public void gaussThreadFinished( )
Implementation of GaussThreadListener interface method.

public void gaussThreadFinished (`int` **actionType**)
Implementation of GaussThreadListener interface method.

public void gaussThreadFinished (`int` **actionType**,
`java.lang.Object[]` **param**)
Implementation of GaussThreadListener interface method.

public void gaussWrite (`java.lang.String` **symbolName**)
NOT THREAD SAVE!!

Write GaussData object from this GaussPanel to Gauss.

public void gaussWriteArray (`java.lang.String[]` **symbolName**)
NOT THREAD SAVE!!

Write GaussData object from this GaussPanel to Gauss.

public void gaussWriteArray (`java.util.Vector` **vec**)
NOT THREAD SAVE!!

Write GaussData object from this GaussPanel to Gauss.

public void gaussWriteArrayLocal (`gauss.GaussData[]` **d**)
NOT THREAD SAVE!!

Write GaussData object from this GaussPanel to Gauss.

public void gaussWriteArrayLocal (`java.util.Vector` **vec**)
NOT THREAD SAVE!!

Write GaussData object from this GaussPanel to Gauss.

public void gaussWriteLocal (`gauss.GaussData` **newData**)
NOT THREAD SAVE!!

Write GaussData object from this GaussPanel to Gauss.

public final Gauss getGauss( )
Returns reference to attached gauss.Gauss Object or null

public GaussData getGaussData (`java.lang.String` **name**)
Returns GaussData from the contextual correct GaussSymbolTable

public final GaussSymbolTable getGaussSymbolTable( )
Returns reference to gauss.GaussSymbolTable this GaussPanel. is attached to or null

public void loadDLibrary (`java.lang.String` **dlibrary**)
Loads Dll to Gauss

public void procedureCallFinished (`util.ProcedureCallEvent` **e**)
Implementation of interface ProcedureCallListener.

public void procedureCallProgress (`util.ProcedureCallEvent` **e**)
Implementation of interface ProcedureCallListener.

public void procedureCallRegistered (`util.ProcedureCallEvent` **e**)
Implementation of interface ProcedureCallListener.

public void procedureCallStarted (`util.ProcedureCallEvent` **e**)
Implementation of interface ProcedureCallListener.

public void propertyChange (`java.beans.PropertyChangeEvent` **evt**)

public void removeGaussData (`java.lang.String` **data**)
Removes GaussData from the contextual correct GaussSymbolTable

public void setGauss (`gauss.Gauss` **g**)

public void setGaussData (`gauss.GaussData` **symbol**)

public final void setGaussSymbolTable (`gauss.GaussSymbolTable` **t**)

public void setVisible (`boolean` **aFlag**)

public void storeData (`double[][]` **data**,
`java.lang.String` **name**)

public void storeData (`double[]` **data**,
`java.lang.String` **name**)

public void storeData (`double` **data**,
`java.lang.String` **name**)

public void storeData (`int[][]` **data**,
`java.lang.String` **name**)

public void storeData (`int[]` **data**,
`java.lang.String` **name**)

public void storeData (`int` **data**,
`java.lang.String` **name**)

public void storeData (`java.lang.String[][]` **data**,
`java.lang.String` **name**)

public void storeData (`java.lang.String[]` **data**,
`java.lang.String` **name**)

public void storeData (`java.lang.String` **data**,
`java.lang.String` **name**)

public void vetoableChange (`java.beans.PropertyChangeEvent` **evt**)

Class **GaussProcedureCall**

Class for defining Gauss computations as a new thread that runs in addition to other Java threads.

Declaration

public abstract class GaussProcedureCall **extends** util.ProcedureCall

Constructors

public GaussProcedureCall( )

Methods

public final void setAllowForMissingValues
(`boolean` **newAllowForMissingValues**)
This property sets the behaviour of communication missing values to Gauss.

public void setGauss (`gauss.Gauss` **gauss**)
Set the Gauss instance for executing the defined task.

Class **GaussSymbolTable**

A GaussSymbolTable serves as a stack for GaussData objects that are shared among objects.
There are the following data access methods:

1. defineGaussData(),
2. getGaussData(),
3. removeGaussData(),
4. setGaussData()

If the argument is not already a key in the HashMap defineGaussData() calls getGaussData() in order to put a new entry in the HashMap. As in GAUSS, GaussData-names defined to be case insensitive.
The getGaussData() method returns the GaussData object mapped to the argument. If no GaussData object is mapped it creates a new (uninitialized) GaussData object, maps, and returns it. If if the argument is `null`, `null` is returned.
The setGaussData() methods do not insert new GaussData objects in the `map` variable. They call either a more general setGaussData() method or the getGaussData() method.

Declaration

public class GaussSymbolTable **extends** java.lang.Object

Constructors

public GaussSymbolTable( )

Methods

public void addPropertyChangeListener
(`java.beans.PropertyChangeListener` **l**)
Add a property change listener to this object.

public void addPropertyChangeListener
(`java.lang.String` **symbolName**,
`java.beans.PropertyChangeListener` **l**)
Add a property change listener to a specific gauss symbol in the variable set.

public boolean containsSymbol (`java.lang.String` **arg**)
Returns true if argument name is already contained in this GaussSymbolTable. GaussData-names defined to be case insensitive.

public String getDescription (`java.lang.String` **name**)
Returns description to specified GaussData. GaussData-names defined to be case insensitive.

public GaussData getGaussData (`java.lang.String` **name**)
Returns reference to GaussData object specified by the name argument. If the argument is a valid gauss symbol name this method guaranties to return a non `null` value. GaussData-names defined to be case insensitive.

public String getName( )
Returns name of this SymbolTable.

public String getSymbolNames( )
Returns all symbol names of this GaussSymbolTable.

public void removeGaussData (`java.lang.String` **symbolName**)
Removes GaussData from this SymbolTable. GaussData-names defined to be case insensitive.

public void removePropertyChangeListener
(`java.beans.PropertyChangeListener` **l**)
Method to remove a property change listener.

public void removePropertyChangeListener
(`java.lang.String` **symbolName**,
`java.beans.PropertyChangeListener` **l**)
Remove property change listener for a specific gauss symbol.

public void resetGaussData (`java.lang.String` **symbolName**)
Clears GaussData.

public void setDescription (`java.lang.String` **description**,
`java.lang.String` **name**)

Sets description to specified GaussData.

public void setGaussData (`double[][]` **data**,
`java.lang.String` **name**)

public void setGaussData (`double[]` **data**,
`java.lang.String` **name**)

public void setGaussData (`double` **data**,
`java.lang.String` **name**)

public void setGaussData (`gauss.GaussData` **newGaussData**)
This is the method to store `GaussData` in the GaussSymbolTable.

public void setGaussData (`gauss.GaussData[]` **newGaussData**)
This is the method to store `GaussData` in the GaussSymbolTable.

public void setGaussData (`gauss.GaussData` **newGaussData**,
`java.lang.String` **name**)

public void setGaussData (`int[][]` **data**,
`java.lang.String` **name**)

public void setGaussData (`int[]` **data**,
`java.lang.String` **name**)

public void setGaussData (`int` **data**,
`java.lang.String` **name**)

public void setGaussData (`java.lang.String[][]` **data**,
`java.lang.String` **name**)

public void setGaussData (`java.lang.String[]` **data**,
`java.lang.String` **name**)

public void setGaussData (`java.lang.String` **data**,
`java.lang.String` **name**)

public void setGaussDataArray (`gauss.GaussData[]` **newGaussData**,
`java.lang.String[]` **name**)

public void setName (`java.lang.String` **newName**)
Sets name to this GaussSymbolTable.

Class **GaussThread**

Class for defining Gauss computations.
Used by GaussPanel.

Declaration

public class GaussThread **extends** java.lang.Thread

Constructors

public GaussThread (`java.lang.String` **name**,
`gauss.GaussSymbolTable` **gaussSymbolTable**,
`java.util.Vector` **commandStack**)

Methods

public void addGaussThreadListener (`gauss.GaussThreadListener` **l**)
Add a listener to the list that's notified each the GaussThread makes noteable stuff.

public void fireGaussThreadFinished( )
Forward the given notification event to all GaussThreadListeners that registered themselves as listeners for a GaussThreadEvent.

public int getActionType( )

public Object getParam( )

public void removeGaussThreadListener (`gauss.GaussThreadListener` **l**)
Remove a listener from the list that's notified each time a notable event in the GaussThread happens.

public void run( )
This method works off the command stack. If a particular command was not successfully executed the thread is terminated immediately leaving the remaining commands from the stack unexecuted. A notice appears in a separate window.

public void setActionType (`int` **newActionType**)

public void setGauss (`gauss.Gauss` **gauss**)
Provide the GaussThread with reference of a GAUSS program.

public void setParam (`java.lang.Object[]` **newParam**)

public void start( )
Before calling the super.start() method a check on null assignments is done.

Class **LocalGaussPanel**

A GaussPanel that references a local GaussSymbolTable.

Declaration

public class LocalGaussPanel **extends** gauss.GaussPanel
implements GaussLocalSymbolRoot

Constructors

public LocalGaussPanel( )

public LocalGaussPanel (`boolean` **isDoubleBuffered**)

public LocalGaussPanel (`java.awt.LayoutManager` **layout**)

public LocalGaussPanel (`java.awt.LayoutManager` **layout**,
`boolean` **isDoubleBuffered**)

Methods

public void addNotify( )
Links this instance to the tree of GaussSymbolTables.

public GaussData getLocalGaussData (`java.lang.String` **name**)

public GaussSymbolTable getLocalGaussSymbolTable( )

public void setLocalGaussData (`gauss.GaussData` **symbol**)

public void storeLocalData (`double` **data**,
`java.lang.String` **name**)

public void storeLocalData (`int` **data**,
`java.lang.String` **name**)

public void storeLocalData (`java.lang.String` **data**,
`java.lang.String` **name**)

Class **Matrix**

Collection of matrix (two dimensional array) manipulation operations.

Declaration

public class Matrix **extends** java.lang.Object

METHODS

public static double add (`double[][]` **arg1**,
`double` **arg2**)

public static double add (`gauss.GaussData` **arg1**,
`double` **arg2**)

public static double add (`gauss.GaussData` **arg1**,
`gauss.GaussData` **arg2**)

public static String append (`java.lang.String` **string**,
`double[][]` **idx**)

public static String append (`java.lang.String` **string**,
`java.lang.String[][]` **arg**)

public static double appendColumn (`double[][]` **target**,
`double[]` **source**)

public static double appendDoubleColumns (`double[][]` **target**,
`double[][]` **source**)

public static double appendDoubleRows (`double[][]` **target**,
`double[][]` **source**)

public static String appendRow (`java.lang.String[][]` **target**,
`java.lang.String[]` **source**)

public static String appendRows (`java.lang.String[][]` **target**,
`java.lang.String[][]` **source**)

public static String appendStringColumns (`java.lang.String[][]` **target**,
`java.lang.String[][]` **source**)

public static String appendStringRows (`java.lang.String[][]` **target**,
`java.lang.String[][]` **source**)

public static double cloneDoubleArray (`double[][]` **x**)

public static double cloneIntArray (`int[][]` **x**)

public static String cloneStringArray (`java.lang.String[][]` **x**)

public static void compareDoubleArrayArg (`double[][]` **arg1**,
`double` **arg2**)

public static boolean compareDoubleArrays (`double[][]` **arg1**,
`double[][]` **arg2**)

public static boolean compareDoubleArrays1D (`double[]` **arg1**,
`double[]` **arg2**)

public static boolean compareIntArrays1D (`int[]` **arg1**,
`int[]` **arg2**)

public static boolean compareStringArrays (`java.lang.String[][]` **arg1**,
`java.lang.String[][]` **arg2**)

public static String concatenate (`java.lang.String[][]` **arg**,
`java.lang.String` **string**)

public static String concatenate (`java.lang.String` **string**,
`java.lang.String[][]` **arg**)

public static double diag (`double[][]` **arg1**,
`double[]` **arg2**)
Inserts arg2 as diagonal into arg1 and returns result.

public static double doubleColumnToMatrix (`double[]` **v**)

public static int doubleToIntArray (`double[]` **dA**)
Converts an one dimensional array of integers to a one dimensional array of doubles.

public static double extractDoubleCol (`double[][]` **matrix**,
`int` **index**)

public static double extractDoubleCols (`double[][]` **matrix**,
`int` **startIndex**,
`int` **endIndex**)

public static double extractDoubleRow (`double[][]` **matrix**,
`int` **index**)

public static double extractDoubleRows (`double[][]` **matrix**,
`int` **startIndex**,
`int` **endIndex**)

public static String extractStringRow (`java.lang.String[][]` **matrix**,
`int` **index**)

public static String extractStringRows (`java.lang.String[][]` **matrix**,
`int` **startIndex**,
`int` **endIndex**)

public static double eye (`int` **r**)
Returns r-dimensional identity matrix.

public static double getDoubleColumn (`double[][]` **matrix**,
`int` **index**)

public static double getDoubleColumns (`double[][]` **matrix**,
`int` **startIndex**,
`int` **endIndex**)

public static double getDoubleRowsArg (`double[][]` **matrix**,
`int` **startIndex**,
`int` **endIndex**)

public static int getIndexOfElement (`java.lang.String` **what**,
`java.lang.String[][]` **where**,
`int` **startRowIndex**,
`int` **startColumnIndex**)

Returns the index of the first ocurrence of the argument in this GaussData object, starting the search at the specified row and column index.
The search is conducted row-wise, i.e. the row at startRowIndex is searched first, then the row at startRowIndex + 1, etc. If the search algorithm finds a matrix element such that m[i][j]==d, the indices i and j are returned as an array of integers with first the row index (i) and then the column index (j). Note that indices start at 0 and end at getRows()-1 / getCols()-1. If the search fails or the GaussData object represents a matrix, int[] {-1,-1} is returned.
This method should be called from the event dispatching thread. See javax.swing.SwingUtilities.invokeLater().

public static String getStringColumn (`java.lang.String[][]` **matrix**,
`int` **index**)

public static double intColumnToMatrix (`int[]` **v**)

public static double intToDoubleArray (`int[]` **iA**)

Converts an one dimensional array of integers to a one dimensional array of doubles.

public static double intToDoubleArray (`int[][]` **iA**)

Converts an two dimensional array of integers to a two dimensional array of doubles.

public static void multiplyArg (`double[][]` **arg1**,
`double` **arg2**)

public static double ones (`int` **r**,
`int` **c**)

Returns a (r,c)-matrix filled with ones.

public static double reshapeDoubleArray (`double[][]` **arg**,
`int` **newRow**,
`int` **newCol**)

public static double rndu (`int` **r**,
`int` **c**)

Returns a (r,c)-matrix filled with uniform[0,1) random numbers.

public static double seqa (`int` **start**,
`int` **increment**,
`int` **n**)
Returns a (n,1)-matrix filled with a additive sequence starting with *start* and increment *increment*.

public static String stringColumnToMatrix (`java.lang.String[]` **arg**)

public static String stringRowToMatrix (`java.lang.String[]` **arg**)

public static double subtract (`gauss.GaussData` **arg1**,
`gauss.GaussData` **arg2**)

public static double toDoubleMatrix (`double[]` **v**)
Transfers the n elements of the input argument into a (n,1)-array.

public static String toStringMatrix (`java.lang.String[]` **s**)
Transfers the n elements of the input argument into a (n,1)-array.

public static double zeros (`int` **r**,
`int` **c**)
Returns a (r,c)-matrix filled with zeros.

Class **StructureChangeEvent**

Event that nofifies about changes in GaussData objects.

Declaration

public class StructureChangeEvent
extends java.beans.PropertyChangeEvent

Constructors

public StructureChangeEvent (`javax.swing.table.TableModel` **source**,
`gauss.GaussData` **oldValue**,
`gauss.GaussData` **newValue**)

Class **SymbolObject**

Provides logical structure of symbol names and a GaussSymbolTable.

Author: Markus Krätzig

DECLARATION

public class SymbolObject **extends** java.lang.Object

CONSTRUCTORS

public SymbolObject( )

public SymbolObject (`java.lang.String[]` **symbolNames**,
`gauss.GaussSymbolTable` **symbolTable**)

public SymbolObject (`java.lang.String` **symbolName**,
`gauss.GaussSymbolTable` **symbolTable**)

METHODS

public String getName( )

public String getNameArray( )

public GaussSymbolTable getTable( )

B.6 Package util.component

B.6.1 Package Contents

Interfaces

Classes

B.6.2 Interfaces

Interface **CardChange**

Defines convenience methods for managing the succession of components in a CardChangePanel.
The CardChangePanel informs the currently displayed component when switching to a new component if the current component implements this interface. The implementator component may veto this switch which interrupts the change of the displayed components.

Declaration

public interface CardChange

Methods

public void setNextCard (`java.awt.Component` **nextCard**)

The implementator can veto in this method that the argument component is displayed as a top level component. The shifts must be managed by a CardChangePanel.

Interface **CardChangeInput**

Defines convenience methods for managing the succession of components in a CardPanel.
The CardChangePanel informs the component displayed just before it is made visible by calling the method `initialSelection()`. This gives the implementator component the chance to synchronize its status with some data base. When made invisible (e.g. the CardChangePanel switches to another component) the `evaluateInput()` method is called which gives the chance to set the input, selections etc. to some data base.

Declaration

public interface CardChangeInput

Methods

public void evaluateInput( )

Called by CardChangePanel when the implementator component is made invisible.

public void initialSelection( )
Called by CardChangePanel just befor the implementator component is made visible.

Interface **SplashScreen**

Implementators are recognized by the splash screen manager. The splash screen manager controls typical tasks a splash screen usually does.

Declaration

public interface SplashScreen

Methods

public void addMessageText (`java.lang.String` **text**)
Add a text increment to an existing message text. (E.g. ". ")

public void dispose( )
Remove the component from the computer screen. Windows and frames should be disposed when calling this method.

public void setMessageText (`java.lang.String` **text**)
Set some message text.

B.6.3 Classes

Class **AngularBrackets**

Matrix-like border for BorderTables and GaussDataTables. Setting this border to a GaussDataTable creates a matrix like object on the screen.
Do not set this border on a JTable (Bug 4222732).

Declaration

public class AngularBrackets **extends** javax.swing.border.AbstractBorder

Constructors

public AngularBrackets( )
Creates a AngularBrackets-border instance.

METHODS

public Insets getBorderInsets (`java.awt.Component` **c**)
Returns the value of getBorderMargins. *Overrides super method.*

public Insets getBorderInsets (`java.awt.Component` **c**,
`java.awt.Insets` **insets**)
Reinitialize the insets parameter with this Border's current Insets. *Overrides super method.*

public boolean isBorderOpaque( )
This default implementation returns false. *Overrides super method.*

public void paintBorder (`java.awt.Component` **c**,
`java.awt.Graphics` **g**,
`int` **x**,
`int` **y**,
`int` **width**,
`int` **height**)
Called by Swing to paint border. **Overrides super method.**

CLASS **BorderTable**

A subclass of JTable that respects borders. JTables do not respect borders at all. This results in Bug 4222732. This BorderTable is a workaround for Java 1.2 and 1.3 In addition to respecting borders the BorderTable can also deal with dynamic column width, e.g. it is possible to configure a BorderTable object such that it automatically adjusts the column width when the data displayed changes. These adjustments can be limited to a minimum and maximum width.
The property `highlightingSelectedCell` can be set to `true` or `false` in order to switch on/ off the frame that appears around a selected table cell. Use the accessor methods `setHighlightingSelectedCell()` to do so.

Implementation remarks:

When resizing columns:

The *delta* which is computed in `sizeColumnsToFit(int)` must be corrected for insets.left and insets.right. We cannot do this in `sizeColumnsToFit(int)` since it calls `accomodateDelta()` which is private. We therefore do it in `accomodateDelta()`. This implies to exactly redefine (copy) the private interfaces Resizable2 and Resizable3, the private methods adjustSizes(long, Resizable2, boolean), adjustSizes(long, Resizable3, boolean), and setWidthFromPreferredWidths(boolean).

Adding a BorderTable to a scroll pane

If a border is set to BorderTable and it is added to a scroll pane this border is transfered from the BorderTable to the scroll pane. This happens in the `addNotify()` method of this class. The reverse is not implemented.
This table draws grid lines before the first row and the first column. For that it needs 1 pixel each direction.

Declaration

public class BorderTable **extends** javax.swing.JTable

Constructors

public BorderTable( )

public BorderTable (`int` **numRows**,
`int` **numColumns**)

public BorderTable (`java.lang.Object[][]` **rowData**,
`java.lang.Object[]` **columnNames**)

public BorderTable (`javax.swing.table.TableModel` **dm**)

public BorderTable (`javax.swing.table.TableModel` **dm**,
`javax.swing.table.TableColumnModel` **cm**)

public BorderTable (`javax.swing.table.TableModel` **dm**,
`javax.swing.table.TableColumnModel` **cm**,
`javax.swing.ListSelectionModel` **sm**)

public BorderTable (`java.util.Vector` **rowData**,
`java.util.Vector` **columnNames**)

Methods

public int columnAtPoint (`java.awt.Point` **point**)
Reimplements super method such that it considers the border now.

public void createDefaultColumnsFromModel( )
Reimplements super method. It regards possible individual column width now.

public Rectangle getCellRect (`int` **row**,
`int` **column**,
`boolean` **includeSpacing**)
Reimplements super method such that it considers the border now.

public int getColumnWidth( )
Returns current column width. The width is *not* computed when calling this method.

public int getMaximumColumnWidth( )
Returns maximum column width property.

public Dimension getMaximumSize( )
Returns the value of a `getPreferredSize()` method call.

public int getMinimumColumnWidth( )
Returns minimum column width property.

public Dimension getMinimumSize( )
Returns the value of a `getPreferredSize()` method call.

public Dimension getPreferredSize( )
Reimplements super method such that it considers the border now.

public String getUIClassID( )
Returns the name of the L&F class that renders this component. The name depends on the version of the JVM running. For Java 1.2 it is `BorderTableUI` otherwise it is the return value of the super method.

public boolean isDynamicColumnWidth( )
Return true if the BorderTable's column width (and the BorderTable's width) depends on the data displayed.

public boolean isHighlightingSelectedCell( )
Return true if the selected cell is highlighted (the default for JTable)

public int maxCellLength( )
Computes the maximum length of all cells. This means that this method evaluates the whole data model. This can be time consuming.

public Component prepareRenderer
(`javax.swing.table.TableCellRenderer` **renderer**,
`int` **row**,
`int` **column**)
Reimplements super method such that it considers
`isHighlightingSelectedCell()` now.

public int rowAtPoint (`java.awt.Point` **point**)
Reimplements super method such that it considers the border now.

public void setBorder (`javax.swing.border.Border` **border**)
Reimplements super method such that the `borderTableBorder` is always set as the most inner border.

public void setColumnWidth (`int` **newValue**)
This method sets the column width of the BorderTable to newValue. If

newValue exceeds the minimum or maximum width, it's adjusted to the appropriate limiting value. Posts a vetoable property change notification with the name "columnWidth".

public void setDynamicColumnWidth (`boolean` **newDynamicColumnWidth**)
Set if the BorderTable's column width should adjust when the data displayed. change. The adjustment takes place within the minimum and maximum width.

public void setHighlightingSelectedCell (`boolean` **newHighlightingSelectedCell**)
Selected cell highlighting method.

public void setIntercellSpacing (`java.awt.Dimension` **newSpacing**)
Sets the width and height between cells to *newSpacing* and redisplays the receiver. Posts a vetoable property change notification with the name "intercellSpacing".

public void setMaximumColumnWidth (`int` **newMaximumColumnWidth**)
Sets the BorderTable's maximum column width to newMaximumColumnWidth.

public void setMinimumColumnWidth (`int` **newMinimumColumnWidth**)
Sets the BorderTable's minimu column width to newMinimumColumnWidth.

public final void sizeColumnsToFit (`int` **c**)
Empty method. Since sizing of BorderTables is determinied by the column width there is no need to fit the columns to set bounds.

public void tableChanged (`javax.swing.event.TableModelEvent` **e**)
Calls super method and determines and sets new maximum column width if `isDynamicColumnWidth()` returns `true`.

Class **BorderTableBorder**

Most inner border for BorderTables. A BorderTable always has an instance of this border set. Event when setting a new border it effectively creates a combined border with an instance of this border as the inner border.
This class is not made public since it is intended to be used solely by the BorderTable.

Declaration

public class BorderTableBorder **extends** javax.swing.border.AbstractBorder

Constructors

public BorderTableBorder( )

Methods

public Insets getBorderInsets (`java.awt.Component` **c**)

public Insets getBorderInsets (`java.awt.Component` **c**,
`java.awt.Insets` **insets**)

public boolean isBorderOpaque( )

public void paintBorder (`java.awt.Component` **c**,
`java.awt.Graphics` **g**,
`int` **x**,
`int` **y**,
`int` **width**,
`int` **height**)

Class **BorderTableUI**

UI implementation for BorderTables when running with Java 1.2.
The paint method of this component ui is modified such that painting the component does not conflict with painting the border of the component.

Declaration

public class BorderTableUI **extends** javax.swing.plaf.basic.BasicTableUI

Constructors

public BorderTableUI( )

Methods

public static ComponentUI createUI (`javax.swing.JComponent` **c**)

public void paint (`java.awt.Graphics` **g**,
`javax.swing.JComponent` **c**)
Reimplements super method such that it considers the border now.

Class **CardChangeInputPanel**

Convenience class for use with CardChange.

Declaration

public class CardChangeInputPanel
extends util.component.CardChangePanel
implements CardChangeInput

Constructors

public CardChangeInputPanel( )

public CardChangeInputPanel (`boolean` **isDoubleBuffered**)

public CardChangeInputPanel (`java.awt.LayoutManager` **layout**)

public CardChangeInputPanel (`java.awt.LayoutManager` **layout**,
`boolean` **isDoubleBuffered**)

Methods

public void evaluateInput( )
The standard implementation calls this method on child components that implement the `CardChangeInput` interface

public void initialSelection( )
The standard implementation calls this method on child components that implement the `CardChangeInput` interface

Class **CardChangePanel**

Convenience class for use with CardChange.

Declaration

public class CardChangePanel **extends** javax.swing.JPanel
implements CardChange

Constructors

public CardChangePanel( )

public CardChangePanel (`boolean` **isDoubleBuffered**)

public CardChangePanel (`java.awt.LayoutManager` **layout**)

public CardChangePanel (`java.awt.LayoutManager` **layout**,
`boolean` **isDoubleBuffered**)

Methods

public Dimension getMaximumSize( )

public Component getNextCard( )

public boolean isFirstTimeShown( )

public boolean isMaximumSize_X_Axis( )

public boolean isMaximumSize_Y_Axis( )

public void setFirstTimeShown (`boolean` **newFirstTimeShown**)

public void setLayout (`java.awt.LayoutManager` **mgr**)
Sets the layout manager for this container.

public void setMaximumSize_X_Axis
(`boolean` **newMaximumSize_X_Axis**)

public void setMaximumSize_Y_Axis
(`boolean` **newMaximumSize_Y_Axis**)

public void setNextCard (`java.awt.Component` **nextCard**)

Class **CardPanel**

A simpler alternative to a JPanel with a CardLayout. The AWT CardLayout layout manager can be inconvenient to use because the special "stack of cards"operations it supports require a cast to use. For example to show the card named "myCard"given a JPanel with a CardLayout one would write:

```
((CardLayout)(myJPanel.getLayout())).show(myJPanel, "myCard");
```

This doesn't work well with Swing - all of the CardLayout display operations, like `show` call validate directly. Swing supports automatic validation (see JComponent.revalidate()); this direct call to validate is inefficient.
The CardPane JPanel subclass is intended to support a layout with a modest number of cards, on the order of 100 or less. A cards name is it's component name, as in java.awt.Component.getName(), which is set when the component is added to the CardPanel:

```
myCardPanel.add(myChild, "MyChildName");
myChild.getName()    "MyChildName"
```

As with CardLayout, the first child added to a CardPanel is made visible and there's only one child visible at a time. The `showCard` method accepts either a childs name or the child itself:

```
myCardPanel.show("MyChildName");
myCardPanel.show(myChild);
```

The CardPanel class doesn't support the vgap/hgap CardLayout properties since one can add a Border, see JComponent.setBorder().
Originallly the `getVisibleChild()` method was private. I changed it to protected. Alexander Benkwitz Feb 2002.
Source: http://java.sun.com/products/jfc/tsc/articles/cardpanel/

Declaration

public class CardPanel **extends** javax.swing.JPanel

Constructors

public CardPanel( )

Creates a CardPanel. Children, called "cards"in this API, should be added with add(). The first child we be made visible, subsequent children will be hidden. To show a card, use one of the show*Card methods.

Methods

public void showCard (`java.awt.Component` **card**)

Hide the currently visible child "card"and show the specified card. If the specified card isn't a child of the CardPanel then we add it here.

public void showCard (`java.lang.String` **name**)

Show the card with the specified name.

public void showFirstCard( )

Show the first card that was added to this CardPanel.

public void showLastCard( )

Show the last card that was added to this CardPanel.

public void showNextCard( )

Show the card that was added to this CardPanel after the currently visible card. If the currently visible card was added last, then show the first card.

public void showPreviousCard( )

Show the card that was added to this CardPanel before the currently visible card. If the currently visible card was added first, then show the last card.

Class **CardPanelAction**

Default implementation for task of showing a "card"component in a CardSuccessionPanel.
The "card"component is constructed when `getComponent()` is called the first time. This shifts possibly time consuming gui construction work from constructing the action object to executing this action.

Declaration

public class CardPanelAction **extends** javax.swing.AbstractAction

Constructors

public CardPanelAction (`javax.swing.JComponent` **what**,
`util.component.CardSuccessionPanel` **where**)

public CardPanelAction (`javax.swing.JComponent` **what**,
`util.component.CardSuccessionPanel` **where**,
`boolean` **enabled**)

public CardPanelAction (`javax.swing.JComponent` **what**,
`util.component.CardSuccessionPanel` **where**,
`boolean` **enabled**,
`java.lang.String` **name**)

public CardPanelAction (`javax.swing.JComponent` **what**,
`util.component.CardSuccessionPanel` **where**,
`boolean` **enabled**,
`java.lang.String` **name**,
`javax.swing.Icon` **icon**)

public CardPanelAction (`javax.swing.JComponent` **what**,
`util.component.CardSuccessionPanel` **where**,
`java.lang.String` **name**)

public CardPanelAction (`javax.swing.JComponent` **what**,
`util.component.CardSuccessionPanel` **where**,
`java.lang.String` **name**,
`javax.swing.Icon` **icon**)

public CardPanelAction (`java.lang.String` **className**,
`util.component.CardSuccessionPanel` **where**,
`boolean` **enabled**,
`java.lang.String` **name**,
`javax.swing.Icon` **icon**)

public CardPanelAction (`java.lang.String` **className**,
`util.component.CardSuccessionPanel` **where**,
`java.lang.String` **name**)

Methods

public void actionPerformed (`java.awt.event.ActionEvent` **arg1**)

public void addComponentToHierarchy( )
Add the component to the CardPanel but does not bring it to the front.

public JComponent getComponent( )
Returns the "card"component. This may be just created when this method is the first time called.

public boolean isComponentInHierarchy( )
Returns true if the "card"component is already in the component hierarchy, otherwise false. The implication of returning true is that the "card"component is already constructed and any operations can be executed on it.

Class **CardSuccessionPanel**

A simpler alternative to a JPanel with a CardLayout.
The behaviour of this class is identical to the superclass but the process of making cards visible and invisible is safeguarded by firering various vetoable property changes. Therefore, the process of hiding the current card and/ or showing a new card can be vetoed by the cards themselfs when certain conditions are not met (not all inputs complete) or a succession is not allowed (i.e. go from A to C over B, but not from A to C directly).

Declaration

public class CardSuccessionPanel **extends** util.component.CardPanel

Fields

public static final String CARD_DISPLAYED

Constructors

public CardSuccessionPanel( )

Methods

public Component getCurrentCard( )

Returns the currently visible child "card". The return value can be `null`.

public void showCard (`java.awt.Component` **card**)

Reimplements the super method. The behaviour is basically the same but the process of making cards visible and invisible is safeguarded by firering various vetoable property changes. Therefore, the process of hiding the current card and/ or showing a new card can be vetoed by the cards themselfs when certain conditions are not met (not all inputs complete) or a succession is not allowed (i.e. go from A to C over B, but not from A to C directly).

Class **InputValidatingTextField**

Performs input validation before input focus is permanently moved away.

Declaration

public abstract class InputValidatingTextField
extends javax.swing.JTextField

Constructors

public InputValidatingTextField( )

public InputValidatingTextField (`javax.swing.text.Document` **doc**,
`java.lang.String` **text**,
`int` **columns**)

public InputValidatingTextField (`int` **columns**)

public InputValidatingTextField (`java.lang.String` **text**)

public InputValidatingTextField (`java.lang.String` **text**,
`int` **columns**)

Methods

public abstract void message( )

Called by the validation mechanism if an event is suitable to move input focus away from this component BUT `validateInput()` returns *false*. Subclasses must provide an implementation for this method (e.g. beeping, flashing, showing Dialogs, Popups etc.).

public void setDocument (`javax.swing.text.Document` **doc**)

public abstract boolean validateInput( )
Should contain the algorithm for input validation. Subclasses must provide an implementation for this method.

Class **MatrixBorder**

Matrix-like border for BorderTables and GaussDataTables. Setting this border to a GaussDataTable creates a matrix like object on the screen.
Do not set this border on a JTable (Bug 4222732).

Declaration

public class MatrixBorder **extends** javax.swing.border.AbstractBorder

Constructors

public MatrixBorder( )

Methods

public Insets getBorderInsets (`java.awt.Component` **c**)

public Insets getBorderInsets (`java.awt.Component` **c**,
`java.awt.Insets` **insets**)

public boolean isBorderOpaque( )

public void paintBorder (`java.awt.Component` **c**,
`java.awt.Graphics` **g**,
`int` **x**,
`int` **y**,
`int` **width**,
`int` **height**)

Class **ModalTextField**

An implementation of the InputValidatingTextField. This class provides a general validation mechanism. Wrong input is reported by a modal dialog with an informative message. This message explains what is wrong with the input.

Declaration

public class ModalTextField
extends util.component.InputValidatingTextField

Constructors

public ModalTextField( )
: Creates a ModalTextField with precision = 0 and right alignment for the text.

Methods

public int getDataType( )
: Returns data type property or 0 if no data type verification is conducted.

public double getDoubleValue( )
: Tries to parse input text to a double. Throws an exception if the text cannot be parsed.

public int getIntervalType( )
: Returns interval type property or 0 if no interval type is set.

public int getIntValue( )
: Tries to parse input text to an integer. Possible double values are truncated. Throws an exception if the text cannot be parsed.

public double getLowerBound( )
: Returns lower bound property. This is `Double.NEGATIVE_INFINITY` if no lower bound is set.

public int getPrecision( )
: Returns precision property for numeric data. The precision is the number of digits right of decimal point.

public String getResetValue( )
: Returns the reset value. Also for numeric data the reset value is specified as a string.

public double getUpperBound( )
: Returns upper bound property. This is `Double.POSITIVE_INFINITY` if no upper bound is set.

public Verifier getVerifier( )
: Returns the object that conducts verification.

public void message( )
: Pops up a message dialog. Called by the validation mechanism if the event queue processes an event that shifts input focus permanently away from this component and the input text did not pass the verifier.

public void setDataType (`int` **dataType**)

Sets data type property. This can be one of `Verifier.LETTER_STRING`, `Verifier.STRING`, `Verifier.DOUBLE`, or `Verifier.INTEGER`.

public void setEnabled (`boolean` **b**)
Calls super method and `setEditable(b)` and repaints the component.

public void setIntervalType (`int` **newIntervalType**)
Sets interval type property. This can be one of `Verifier.CLOSED` for *I=[a,b]*, `Verifier.OPEN` for *I=(a,b)*, `Verifier.LEFT_OPEN` for *I=(a,b]*, or `Verifier.RIGHT_OPEN` for *I=[a,b)* intervals

public void setLowerBound (`double` **newLowerBound**)
Sets lower bound property.

public void setPrecision (`int` **precision**)
Sets number of digits that should appear right of decimal point.

public void setResetValue (`java.lang.String` **newResetValue**)
Set a new reset value. This text is inserted if a escape key event is processed.

public void setUpperBound (`double` **newUpperBound**)
Sets upper bound property.

public void setVerifier (`util.component.Verifier` **newVerifier**)
Sets a new object that does the verification task.

public boolean validateInput( )
Verifies the input text and sets the `errorString` field to some message if verification fails.

Class **MultiLineLabel**

Covenience label class that can display text with more than one line.
A multi line label text is set with the usual `setText()` method. The lines are separated by `\n`, e.g. `new MultiLineLabel("line1\nline2\nline3")`.

Declaration

public class MultiLineLabel **extends** javax.swing.JLabel

Constructors

public MultiLineLabel( )

public MultiLineLabel (`javax.swing.Icon` **image**)

public MultiLineLabel (`javax.swing.Icon` **image**,
`int` **horizontalAlignment**)

public MultiLineLabel (`java.lang.String` **text**)

public MultiLineLabel (`java.lang.String` **text**,
`javax.swing.Icon` **icon**,
`int` **horizontalAlignment**)

public MultiLineLabel (`java.lang.String` **text**,
`int` **horizontalAlignment**)

Methods

public String getUIClassID( )
Returns *MultiLineLabelUI*.

Class **MultiLineLabelUI**

Author: Zafir Anjum http://codeguru.earthweb.com/java/articles/198.shtml
Use multi-line text in a JLabel. To enable multi-line in a JLabel we have to specify our own LabelUI class that will render the multiple lines. Although the code shown below seems quite long, most of it is actually Swing code being reused.

Declaration

public class MultiLineLabelUI **extends** javax.swing.plaf.basic.BasicLabelUI

Constructors

public MultiLineLabelUI( )

Methods

public static Dimension computeMultiLineDimension
(`java.awt.FontMetrics` **fm**,
`java.lang.String[]` **strs**)

public static String layoutCompoundLabel (`java.awt.FontMetrics` **fm**,
`java.lang.String[]` **text**,
`javax.swing.Icon` **icon**,
`int` **verticalAlignment**,
`int` **horizontalAlignment**,
`int` **verticalTextPosition**,
`int` **horizontalTextPosition**,
`java.awt.Rectangle` **viewR**,
`java.awt.Rectangle` **iconR**,
`java.awt.Rectangle` **textR**,
`int` **textIconGap**)

Compute and return the location of the icons origin, the location of origin of the text baseline, and a possibly clipped version of the compound labels string. Locations are computed relative to the viewR rectangle. This layoutCompoundLabel() does not know how to handle LEADING/TRAILING values in horizontalTextPosition (they will default to RIGHT) and in horizontalAlignment (they will default to CENTER). Use the other version of layoutCompoundLabel() instead.

public static String layoutCompoundLabel (`javax.swing.JComponent` **c**,
`java.awt.FontMetrics` **fm**,
`java.lang.String[]` **text**,
`javax.swing.Icon` **icon**,
`int` **verticalAlignment**,
`int` **horizontalAlignment**,
`int` **verticalTextPosition**,
`int` **horizontalTextPosition**,
`java.awt.Rectangle` **viewR**,
`java.awt.Rectangle` **iconR**,
`java.awt.Rectangle` **textR**,
`int` **textIconGap**)

Compute and return the location of the icons origin, the location of origin of the text baseline, and a possibly clipped version of the compound labels string. Locations are computed relative to the viewR rectangle. The JComponents orientation (LEADING/TRAILING) will also be taken into account and translated into LEFT/RIGHT values accordingly.

public String splitStringByLines (`java.lang.String` **str**)

Class **SplashScreenManager**

A collection of static methods that can be called for performing typical tasks with the splash screen.

Declaration

public class SplashScreenManager **extends** java.lang.Object

Methods

public static void addSplashScreenMessage (`java.lang.String` **text**)
Adds the argument to the message text of the current splash screen.

public static void disposeSplashScreen (`int` **millis**)
Waits millis/1000 seconds and then disposes the splash screen.

public static void setMainFrame (`javax.swing.JFrame` **newMainFrame**)
The splash screen is always displayed in front of the argument frame.

public static void setSplashScreen
(`util.component.SplashScreen` **newSplashScreen**)
Sets a new splash screen, disposes old splash screen, shows new splash screen. The new splash screen can be `null` (e.g. for calling `dispose()` on the current splash screen).

public static void setSplashScreenMessage (`java.lang.String` **text**)
Set a new message text to the current splash screen.

Class **Verifier**

"Filter"object that can be used for checks whether input data matches certain requirements. The most obvious usage is to filter out invalid inputs.
If one wants to verify numeric input data valid types are `InputFilter.DOUBLE` and `InputFilter.INTEGER` and valid classes for the upper and lower bounds are `null` `java.lang.Double` `gauss.GaussData` (of type GaussData.MATRIX).
In case of verifying literal data valid types are `InputFilter.LETTER_STRING` and `InputFilter.STRING` and valid classes for the upper and lower bounds are `null`
If input data is numeric an interval can be specified additionally. This can be one of CLOSED, OPEN, LEFT_OPEN, and RIGTH_OPEN.

If the input data is not numeric the interval type property is not considered when passing arguments through this filter.
If the upper (lower) bound is `null` no bound checking is performed for the upper (lower) bound.

If the upper (lower) bound is non `null` AND not one of the specified class (see VALID CLASSES FOR BOUNDS) no bound checking is performed for the upper (lower) bound and a `gauss.BoundException` thrown.

Declaration

public class Verifier **extends** java.lang.Object

Fields

public static final int CLOSED
This field stores the identifier of a closed interval. Let *x1* be the lower bound, and *x2* be the upper bound of the interval. Using the identifier `CLOSED` means to define the interval *[x1,x2]*.

public static final int OPEN
This field stores the identifier of an open interval. Let *x1* be the lower bound, and *x2* be the upper bound of the interval. Using the identifier `OPEN` means to define the interval *(x1,x2)*.

public static final int LEFT_OPEN
This field stores the identifier of a left open interval. Let *x1* be the lower bound, and *x2* be the upper bound of the interval. Using the identifier `LEFT_OPEN` means to define the interval *(x1,x2]*.

public static final int RIGHT_OPEN
This field stores the identifier of a right open interval. Let *x1* be the lower bound, and *x2* be the upper bound of the interval. Using the identifier `RIGHT_OPEN` means to define the interval *[x1,x2)*.

public static final int LETTER_STRING
Identifier for string input value. This string must start with a letter ('a',..'Z' or '_'.

public static final int STRING
Identifier for general string input value.

public static final int DOUBLE
Identifier for double value.

public static final int INTEGER
Identifier for integer value.

Constructors

public Verifier( )
Constructs default Verifier.
Accept only INTEGER, no bound checking performed.

public Verifier (`double` **lowerBound**,
`double` **upperBound**,
`int` **dataType**,
`int` **intervalType**)
Constructs Verifier and defining the acceptance set by a lower bound, upper bound, data type, and interval type.

public Verifier (`int` **dataType**,
`int` **intervalType**)
Constructs Verifier and defining the acceptance set by data type, and interval type.

public Verifier (`java.lang.Object` **lowerBound**,
`java.lang.Object` **upperBound**,
`int` **dataType**,
`int` **intervalType**)

Constructs Verifier and defining the acceptance set by a lower bound, upper bound, data type, and interval type. For the bound objects instances of java.lang.Number or gauss.GaussData are recognized.

Methods

public boolean contains (`java.lang.String` **text**)

Method to check whether `text` goes through this input filter.
If this filter should accept only numeric values the argument must not contain a *FloatTypeSuffix*. (See §3.10.2 of the Java Language Specification.) This is more restrictive than the lexical rule of a FloatValue which is

Sign(opt) FloatingPointLiteral

where *Sign* and *FloatingPointLiteral* are defined in §3.10.2 of the Java Language Specification. *FloatingPointLiteral* may contain *FloatTypeSuffix* which is one of {f, F, d, D}.
This method throws a `gauss.BoundException` if the upper or lower bound objects can not be interpreted in a meaningful manner.

public int getDataType( )

Returns current data type.
This can be one of DOUBLE, INTEGER, STRING, or LETTER_STRING.

public String getDetailsAcceptanceSet( )

Forms string representation about the acceptable values.

public String getDetailsInputValue( )

Forms string representation about the input value.

public int getIntervalType( )

Returns current interval type.
This can be one of CLOSED, OPEN, LEFT_OPEN, or RIGHT_OPEN.

public String getLastMessage( )

Calling this method after`contains(String text)` returned false gives more detailed information why acceptance was rejected.

public Object getLowerBound( )

Returns the lower bound of this interval.

public Object getUpperBound( )

Returns the upper bound of this interval.

public void setDataType (`int` **newDataType**)
Sets a new data type.
If the argument is invalid the current data type stays unchanged.

public void setIntervalType (`int` **newIntervalType**)
Sets new interval type.
If the argument is invalid the current interval type stays unchanged

public void setLowerBound (`java.lang.Object` **lowerBound**)
Sets the lower bound of this interval.

public void setUpperBound (`java.lang.Object` **upperBound**)
Sets the upper bound of this interval.

public String toString( )
This method returns a string representation of this interval object.

Appendix C

Gauss Control: A software which connects to GAUSS for Windows

The software *Gauss Control* (GC) can be used to control GAUSS automatically. Development of GC was motivated by the need to run GAUSS royalty free as a statistical engine. The tasks solved by GC include the start and termination of GAUSS; make function calls to, run programs in, and stop running programs in GAUSS as well as to write data to, and read data from GAUSS. This appendix documents structure and usage of GC. The **copyright** and the **disclaimer** of Appendix A apply to the software *Gauss Control*.

GC has been tested on Windows 95, 98, 2000 and NT 4.0. with GAUSS for Windows 3.2, 3.5 and 3.6. It is developed by the author (for GAUSS 3.2) and by Markus Krätzig (for GAUSS 3.5 and higher). An archive file containing the binary files and the GAUSS source code can be downloaded at http://www.jstatcom.com/download

C.1 Set up and configuration

Gauss Control consists of two communication libraries. One is loaded by the application (in the following the mother process) that wants to run GAUSS as a computing engine. It is written in *C* and compiled to the dynamic library (DLL) `xlm.dll`. The other is the GAUSS-library `xlm` which is loaded together with the DLL `glm.dll` (also written in *C*) by GAUSS. The GAUSS-library must be built using the GAUSS `lib` function before running GC. Each library represents one side of the communication channel.

C.1.1 File structure

GC consists of the following files:

`gauss32.cfg`	Template configuration file for GAUSS 3.2,
`gauss35.cfg`	Template configuration file for GAUSS 3.5 or higher,
`pqgrun.cfg`	Template configuration file for GAUSS graphics library,
`xlm.cfg`	Configuration file for *Gauss Control*,
`xlm.src`	Source files for GAUSS library `xlm`,
`glm.dll`	DLL-functions used by GAUSS,
`vdlg.dll`	Version dialog library,
`xlm.dll`	DLL-functions used by the mother process.

The configuration files `gauss32.cfg`, `gauss35.cfg`, and `pqgrun.cfg` can be customized or replaced by other valid configuration files. The configuration file `xlm.cfg` contains the configuration variables for GC (see next subsection).

C.1.2 Configuration

In the following it is assumed that the mother process is executed from some directory (working directory) and loads the DLL `xlm.dll`. This working directory is denoted as `%WD%` in the following. In order to execute the functions

of `xlm.dll` the following configuration is assumed.

The configuration file `xlm.cfg` is read in from the *XLM directory* (see below) when `xlm.dll` is loaded by the mother process. It contains the following configuration variables:

Variable	Purpose	Default
`GAUSSEXE`	Full path and name of the GAUSS executable	C:\gauss\gauss.exe
`GAUSSVERSION`	Version of GAUSS	
`SHAREDMEMSIZE`	Size (in bytes) of shared memory	800000
`TEMPPATH`	Path in which the temporary directory `xlmgauss` is created	C:\temp

The configuration file must be customized before starting GC the first time. For that it is necessary to know the name and the full path of the GAUSS executable (`GAUSSEXE`). The variable `GAUSSVERSION` can take either the value `3.2` for GAUSS 3.2 or the value `3.5` for GAUSS 3.5 and higher. If this variable stays empty the user is prompted with a dialog to make a selection the first time GC runs. `SHAREDMEMSIZE` can be increased or decreased depending on the size of data communicated with GC and system resources. The size of the shared memory determines the maximum size of the data matrices (strings and numbers) that can be transferred from the mother process to GAUSS and vice versa. The default of 800,000 bytes is equivalent to a matrix with 100,000 numbers. The transfer of a matrix with more numbers will display an error message. This error is not critical, i.e. the transfer terminates and GC resumes its communication duties. In this case the default should be increased for the future. `TEMPPATH` must be set to an existing directory with read and write access if no `TEMP` or `TMP` environment variables are set. See below how GC finds and uses the temporary directory.

Beside the above configuration variables the software uses environment vari-

ables if they are defined on the local computer system. They are used to determine the following two directories:

XLM directory: The directory containing the files `gauss32.cfg`, `gauss35.cfg`, `pqgrun.cfg`, `xlm.cfg`, `xlm.src`, and `glm.dll`.

1. Environment variable `XLM_PATH`
2. Assume default value `%WD%\xlm`

Temporary directory: The directory where GC creates the (temporary) sub directory `xlmgauss`.

1. Configuration variable `TEMPPATH`
2. Environment variable `TEMP`
3. Environment variable `TMP`

In the following abbreviated as `%TEMPDIR%`.

C.1.3 Temporary files

In general, data is exchanged with shared memory. Beside that, GAUSS and GC use log and temporary files (e.g. for error files or graphic output). In order to use these files exclusively for a specific GAUSS program running they are stored in a special named and created directory. Before GAUSS is started by GC a new temporary directory is created. It is placed in the temporary directory `%TEMPDIR%\xlmgauss\` (e.g. `%TEMPDIR%\xlmgauss\s5l_\`). Usually it is deleted when GAUSS is finished by GC. If it is not deleted automatically the directory `%TEMPDIR%\xlmgauss` and all subsequent subdirectories can be deleted if GC is not running.

C.2 The mother process

In order to use GAUSS as a statistical engine the mother process must load the DLL `xlm.dll`. A full documentation of the exported functions can be found in Section C.6.

First, the mother process should execute the function

`startGauss`.

When this function returns `true` a new instance of GAUSS has been started and is ready to work for the mother process.

Then, the mother process can call any of the following self explaining functions in order to perform computation in GAUSS:

`executeCommand`,

`executeCommands`,

`executeProgram`,

`writeMatrix`,

`writeString`,

`readDataMatrix`.

It is guaranteed that these functions do not return before the requested task on the GAUSS side has been finished. Finishing means that the task has been executed with or without success and GAUSS is ready to receive the next command. GC is fully synchronized with respect to the thread in the mother process that executes one of the above functions and GAUSS. However, if the mother process is a multi threaded application it must make sure that multiple GAUSS jobs (represented by different threads) do not interfere with each other.

If a computation request fails, a more detailed error message is obtained with

`getLastError`.

In order to load dynamic link libraries in GAUSS it is advisable to use

`loadDlibrary`

instead of the respective GAUSS function.

The name of the special created temporary directory is returned by the function

`getTempPath`.

It can be used as a default for storing temporary results.

The mother process can test whether GAUSS executes a program or procedure call and can stop that execution. This is done by calling the function

`isExecuting` and

`stopExecution`.

If the running instance of GAUSS should be terminated the mother process must execute

`terminateGauss`.

If the mother process terminates but wants to leave GAUSS running it should execute

`unlinkGauss`

before it terminates itself. Unlinking GAUSS unsets its the communication abilities.

C.3 The GAUSS side

GAUSS is started by the mother process with the instruction to load the GAUSS-library `xlm` and the DLL `glm.dll`. This library provides a set of communication functions that are used by the mother process for communication with GAUSS. The functions are documented in Section C.6. They should not be used directly in GAUSS code (exception: `showLastGraphic`). GAUSS does not actively write data to the mother process. If data is communicated from GAUSS to the mother process, GAUSS is instructed to do so using the functions and procedures defined in `xlm`.

When GAUSS finished loading the libraries `xlm` and `glm.dll` the set up of the interprocess communication is finished. GAUSS is now completely controlled by the mother process. All function calls and requests to run a GAUSS program come from the mother process now. Data transfer is also initiated by the mother process. There are no functions that allow GAUSS to actively write or read data to or from the mother process.

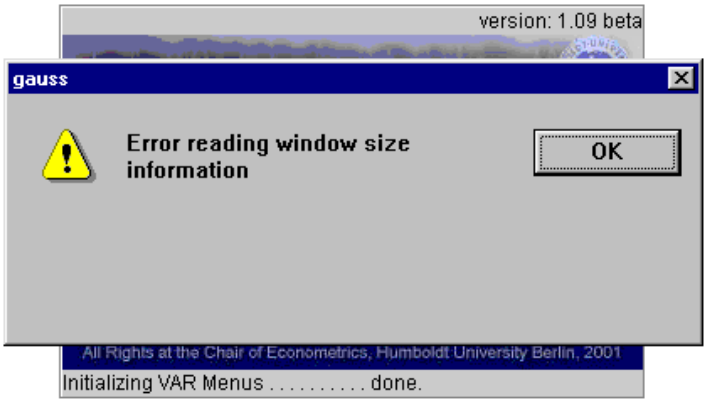


Figure C.1: Warning message at program start when GAUSS 3.2 was quit from a minimized window.

There is a known GAUSS 3.2 warning message that may block the start of the mother process. This bug is related to the inability of GAUSS 3.2 to re–start correctly when it was terminated from a minimized window. In this case the mother process can stop initialization and the error message as shown in Figure C.1 appears. The mother process will resume initialization after clicking the OK–button.

In order to avoid this error with future JMulTi starts, it is advisable to exit JMulTi, start only GAUSS and close it when the GAUSS–application window is not minimized. In general, one should always finish GAUSS from a window that is **not** minimized.

C.4 Example and test software

The Windows console program `TestXLM` is a software that tests and demonstrates GC. The following menu is displayed after startup of `TestXLM`:

```
Menu:
------------------------
Quit                 = 0
Write matrix         = 1
Write string         = 2
Read symbol          = 3
Run test             = 4
Set desktop mode     = 5
Set linked mode      = 6

This menu            = 9
```

The menu provides access to some functionality of GC that can be selected and executed. The option `Run test = 4` demonstrates the speed of GC. Matrices or string arrays are created from random data and written to GAUSS, reread again and then compared with the original data.

C.5 Application with JMulTi

GC is used by the Java library JStatCom (see Appendix B), and thus by JMulTi (see Chapter 3) in order to use GAUSS as a statistical engine for any Java application and for JMulTi. In this case the DLL `jxlm.dll` is loaded by the Java application (instead of `xlm.dll`). `jxlm.dll` contains all so called native methods used by the Java program.

C.6 Documentation of exported functions in `xlm.dll`

C.6.1 Data structure

The data structure `GAUSS_SYMBOL` is defined as

```
struct GAUSS_SYMBOL {
  void *pMatrix;   // points to beginning of matrix array
  int rows;        // number of rows
  int cols;        // number of columns
  int type;        // type identification
  char *error;     // points to possible error message
};
```

Data of this type is returned by the function `readDataMatrix`.

C.6.2 Functions

executeCommand

Purpose

Executes a command line in GAUSS.

Format

```
char* executeCommand(char* command)
```

Input

`char* command` The command line, null terminated string.

Output

`char*` Null if GAUSS executed the command line successfully, otherwise some error message (null terminated string).

Remarks

The argument command line does not need to end with ';'. The function is fully synchronized, i.e. it returns when GAUSS finished the execution of all command lines or if execution is interrupted by some error.

See

executeCommands

Source

`xlm.cpp, xlmx.h`

executeCommands

Purpose

Executes n command lines in GAUSS.

Format

```
char* executeCommands(char** command, int n)
```

Input

`char** command`	Pointer to first (of n) command line (null terminated string).
`int n`	Number of strings in the first argument.

Output

`char*`	Null if GAUSS executed the command lines successfully, otherwise some error message (null terminated string).

Remarks

The n command lines do not need to end with ';'. The function is fully synchronized, i.e. it returns when GAUSS finished the execution of all command lines or if execution is interrupted by some error.

See

executeCommand

Source

`xlm.cpp, xlmx.h`

executeProgram

Purpose

Executes a GAUSS program file.

Format

```
char* executeProgram(char* fileName)
```

Input

`char* fileName`	Null terminated string containing the name of the program file to be executed. GAUSS must be able to find the program file, i.e. it must name the full path or must be located in one of the GAUSS search paths.

Output

`char*`	Null if GAUSS executed the program file successfully, otherwise some error message (null terminated string).

Remarks

The function is fully synchronized, i.e. it returns when GAUSS finished the execution of the argument program file or if execution is interrupted by some error.

Source

`xlm.cpp, xlmx.h`

getLastError

Purpose

Returns last error buffer.

Format

```
char* getLastError()
```

Output

`char*` Null if error buffer is empty, otherwise last error message (null terminated string).

Remarks

The function should be called if `startGauss`, `terminateGauss`, or `unlinkGauss` return `false`. In these cases the last error buffer is filled with information about the cause of the failure.

Source

`xlm.cpp, xlmx.h`

getTempPath

Purpose

Returns the temporary directory used by GAUSS started with `startGauss`.

Format

```
char* getTempPath()
```

Output

`char*` Full name of the temporary directory, or ``null'' if it could not be found.

Source

`xlm.cpp, xlmx.h`

isExecuting

Purpose

Checks whether GAUSS currently executes some command or program file.

Format

```
bool isExecuting()
```

Output

`bool` `true` if GAUSS is busy, otherwise `false`.

Source

`xlm.cpp, xlmx.h`

loadDlibrary

Purpose

Loads a set of DLLs in GAUSS.

Format

```
char* loadDlibrary(char* dlibraryString)
```

Input

`char* dlibraryString`	A comma separated list of DLL names that are loaded by GAUSS (null terminated string).

Output

`char*`	Null if GAUSS could load all DLLs successfully, otherwise some error message (null terminated string).

Remarks

The function is fully synchronized, i.e. it returns when GAUSS finished loading the DLLs or if execution is terminated by some error.

Source

`xlm.cpp, xlmx.h`

readDataMatrix

Purpose

Reads a global symbol from GAUSS.

Format

```
GAUSS_SYMBOL readDataMatrix(char* name)
```

Input

`char* name` Symbol name of the global symbol in GAUSS.

Output

`GAUSS_SYMBOL` Data structure of type `GAUSS_SYMBOL`.

Remarks

`GAUSS_SYMBOL` is defined as:

```
struct GAUSS_SYMBOL {
    void*  pMatrix;  // points to beginning of matrix array
    int    rows;     // number of rows
    int    cols;     // number of columns
    int    type;     // type identification
    char*  error;    // points to possible error message
};
```

If the function fails (i.e. it fails to read the global symbol `name` from GAUSS), `GAUSS_SYMBOL.error` is set to an error message, otherwise this element is `null`.

If data was successfully read from GAUSS `GAUSS_SYMBOL.type` specifies how to interpret `GAUSS_SYMBOL.pMatrix`. If `GAUSS_SYMBOL.type=6` it must be interpreted as `double**`. If `GAUSS_SYMBOL.type=13` (or `15`) it must be interpreted as `char***`.

The function is fully synchronized, i.e. it is guaranteed that all data is communicated from GAUSS if the function returns successfully.

See

writeMatrix

writeString

Source

`xlm.cpp, xlmx.h`

startGauss

Purpose

Starts a new instance of GAUSS.

Format

```
bool startGauss()
```

Output

`bool`	`true` if GAUSS could be started and configured as a statistical engine, otherwise `false`.

Remarks

If this function returns `false` call `getLastError()` to get detailed information about the error.

See

getLastError

terminateGauss

Source

`xlm.cpp, xlmx.h`

stopExecution

Purpose

Stops current execution of any GAUSS code.

Format

```
stopExecution()
```

Remarks

It is valid to call this function even if `isExecuting` returns `false`.

See

isExecuting

Source

```
xlm.cpp, xlmx.h
```

terminateGauss

Purpose

Terminates GAUSS that was started with `startGauss`.

Format

```
bool terminateGauss()
```

Output

`bool` `true` if GAUSS was terminated successfully, otherwise `false`.

Remarks

If this function returns `false` call `getLastError()` to get detailed information about the error.

See

getLastError

startGauss

Source

`xlm.cpp, xlmx.h`

unlinkGauss

Purpose

Disconnects the application that called `startGauss` from GAUSS.

Format

```
bool unlinkGauss()
```

Output

`bool`	`true` if GAUSS was disconnected successfully, otherwise `false`.

Remarks

If GAUSS is unlinked it continues to run in the so called *desktop mode*, i.e. such as it was started from the desktop. If this function returns `false` call `getLastError()` to get detailed information about the error.

See

getLastError

startGauss

Source

`xlm.cpp, xlmx.h`

writeMatrix

Purpose

Write array of numeric data to GAUSS.

Format

```
char*
   writeMatrix(char* name, int rows, int cols, double*
               data[])
```

Input

`char* name`	Symbol name of matrix, null terminated string.
`int rows`	Row number of matrix.
`int cols`	Column number of matrix.
`double* data[]`	Array of pointers, each representing a row of the data matrix.

Output

`char*` Null if the whole matrix was written to GAUSS, otherwise some error message (null terminated string).

Remarks

The numeric data is assigned to the global symbol with the name `name`. If `name` already exists in GAUSS it is overwritten otherwise it is created. The function is fully synchronized, i.e. it is guaranteed that all data is communicated to GAUSS if the function returns `null`.

See

readDataMatrix

writeString

Source

`xlm.cpp, xlmx.h`

writeString

Purpose

Write array of strings to GAUSS.

Format

```
char*
   writeString(char* name, int rows, int cols, char*** data)
```

Input

`char* name`	Symbol name of string array, null terminated string.
`int rows`	Row number of string array.
`int cols`	Column number of string array.
`char***`	Pointer to a structure of `row`×`col` null terminated strings, representing the string array.

Output

`char*` Null if the whole string array was written to GAUSS, otherwise some error message (null terminated string).

Remarks

The string array is assigned to the global symbol with the name `name`. If `name` already exists in GAUSS it is overwritten otherwise it is created. The function is fully synchronized, i.e. it is guaranteed that all data is communicated to GAUSS if the function returns `null`.

See

readDataMatrix

writeMatrix

Source

`xlm.cpp, xlmx.h`

C.7 Documentation of GAUSS-library `xlm`

The software GC defines two modes in which GAUSS can run: the *desktop mode* and the *linked mode*. GAUSS appears in desktop mode if it is started as a normal Windows application. If it is linked to a mother process and serves as a statistical engine it is assumed to run in the linked mode. The GAUSS-library `xlm` defines functions for switching between these modes.

C.7.1 Global variables

The GAUSS library `xlm` defines the following global variables. They must not be used by any other GAUSS procedure or function call and must not be used as input or output parameters in procedure or function calls (see below).

variable name	variable type
`__xlm_r`	matrix
`__xlm_c`	matrix
`__xlm_s`	matrix
`__xlm_t`	matrix
`__xlm_swap_file`	string
`__xlm_swap_var`	matrix

C.7.2 Procedures

desktopMode

Purpose

Switches GAUSS from linked mode to desktop mode.

Format

```
dllcall desktopMode;
```

Remarks

Called when GAUSS is unlinked from the mother application.

Source

```
glm.cpp
```

linkedMode

Purpose

Switches GAUSS from desktop mode to linked mode.

Format

```
dllcall linkedMode;
```

Remarks

Used when GAUSS is started by the mother application with `startGauss`.

Source

`glm.cpp`

quitGauss

Purpose

Terminates GAUSS programmatically.

Format

```
dllcall quitGauss;
```

Remarks

GAUSS is terminated without confirmation. It can be called in linked and desktop mode.

Source

```
glm.cpp
```

readMatrix

Purpose

Reads a matrix from the shared memory.

Format

```
dllcall readMatrix(mat, r, c);
```

Input

`mat`	`(r,c) matrix`	Variable that gets filled with the data
`r`	`(1,1) matrix`	Number of rows
`c`	`(1,1) matrix`	Number of columns

Remarks

Before reading a matrix from the shared memory the memory needed in GAUSS for storing the matrix must be provided with e.g. `mat = zeros(r,c);`.

See

writeMatrix

Source

`glm.cpp`

readString

Purpose

Reads a string from the shared memory.

Format

```
dllcall readString(str, l);
```

Input

`str (1,1) string` String of length `size`. It gets filled with the data.
`l (1,1) matrix` Length of `str`

Remarks

Before reading a string from the shared memory the memory needed in GAUSS for storing the string must be provided with e.g. `str = chrs(zeros(l,1);`.

Used by `readStringArray`.

See

readString

readStringArray

Source

`glm.cpp`

readStringArray

Purpose

Reads a string array from the mother application.

Format

```
strArray = readStringArray(buffer, s, r, c);
```

Input

`buffer`	`(1,1) string`	must have size `s`
`s`	`(1,1) matrix`	buffer size
`r`	`(1,1) matrix`	number of rows
`c`	`(1,1) matrix`	number of cols

Output

`strArray`	`(r,c) string array`	String array containing the data.

Source

`xlm.src`

showLastGraphic

Purpose

Displays the last generated GAUSS graphic in front of any other application window.

Format

```
dllcall showLastGraphic;
```

Remarks

This function should be called whenever some graphic output is generated by a GAUSS procedure. Calling this function ensures that the last graphic window comes to front. It thus notifies the user about computational progress.

If GAUSS 3.5 or higher is used an external viewer is started.

Source

`glm.cpp`

symbolTable

Purpose

Returns a list of all global symbols of type matrix, string, and string array.

Format

```
strArray = symbolTable();
```

Output

`strArray (r,4) string array`	String array containing information about the global symbols. Column 1 = symbol name, Column 2 = rows, Column 3 = columns, Column 4 = data type.

Source

`xlm.src`

writeMatrix

Purpose

Writes a matrix to the shared memory.

Format

```
dllcall writeMatrix(mat, r, c);
```

Input

`mat`	`(r,c) matrix`	Variable that is written to the shared memory
`r`	`(1,1) matrix`	Number of rows of `mat`
`c`	`(1,1) matrix`	Number of columns of `mat`

See

readMatrix

Source

`glm.cpp`

writeString

Purpose

Writes a string to the shared memory.

Format

```
dllcall writeString(str, l);
```

Input

`str`	`(1,1) string`	String of length `l` that is written to the shared memory
`l`	`(1,1) matrix`	Lenghth of `str`

See

readString

Source

`glm.cpp`

writeStringArray

Purpose

Writes a string array to the mother application.

Format

```
writeStringArray(s);
```

Input

`s (M,N) string array` Data that is written.

Remarks

Possible errors (a) symbol `s` is not of type string or string array, (b) the dllcall returns with error. The error message is written to the errorlog file.

Source

`xlm.src`

C.8 Suggestions for future extensions

The following improvements could be considered in a future version of this software:

1. No size restrictions for data transfer. In general it is possible to transfer matrices of any size (of course subject to system resources) with a fixed sized shared memory by implementing some synchronization mechanism. In this case the shared memory would be repeatedly filled and read in one transfer operation.

2. Writing data actively from GAUSS to the mother process, e.g. for receiving information about computational progress in GAUSS.

Appendix D

Documentation of the GAUSS library `var`

The library `var` is a set of procedures for analysing VAR and VEC models. The strengths of this library are the various bootstrap confidence intervals provided with the impulse response analysis. This Appendix documents the implementation in GAUSS and all procedures of `var`.
The following files belong to the GAUSS library `var`:

var.dec	var_3sls.src	var_V4I.src
var_boot.src	var_est.src	var_est1.src
var_est2.src	var_est3.src	var_gls.src
var_init.src	var_ira.src	var_irap.src
var_irbo.src	var_irc1.src	var_irc2.src
var_irc3.src	var_joh.src	var_ls.src
var_mx.src	var_new.src	var_ols.src
var_rest.src	var_show.src	var_tool.src
var_var.src	var_spec.src	var_subs.src

All source files can be found in the `src`-subdirectory of the JMulTi installation

directory, for example in `c:\jmulti\src`.

Each procedure starts with a documentation section. It contains purpose, syntax, input arguments, output arguments, remarks, and cross references to other procedures. Only few specialized auxillary procedures are not documented.

D.1 Implementation in GAUSS

This section outlines how the task of writing a library for bootstrapping impulse responses was solved. It can be skipped if the reader is mainly interested in applying this library to an econometric problem.

The main problems faced were: (a) The duplication of model variables when performing the bootstrap (e.g. a model parameter is mirrored by its bootstrap analogon), (b) The work with more than one model simultaneously (e.g. estimating the underlying model and its bootstrap version), and (c) The admission of very general models.

This was solved by viewing a model as a collection of various data (observations, variable names, estimation methods, restrictions, ...). On those entities were procedures applied (e.g. computing bootstrap confidence intervals for impulse responses). This is advantageous for at least two reasons. Data that belong together are stored together and cannot be confused with other model data by accident. Analysing simple models stays simple since the user is not confronted with a bunch of input arguments that are not used for simple models.

A collection of model data is simply a Gauss data buffer. A Gauss data buffer is stored in a variable. Data is added to an existing data buffer with the Gauss command `dbufNew = vput(dbuf,someData,aName)`. The argument `someData` is now an element of the data buffer `dbufNew` with the element name `aName`. The element name uniquely identifies an element. Many proce-

dures take a data buffer as an argument or return a data buffer. The can be viewed and manipulated at any time. They have of course special meanings which are explained in the subsequent section of this chapter.
The advantage is that only one argument (the data buffer `dbuf`) is supplied to functions. The software keeps well structured and is easy to maintain although complex models and complex operations are implemented. The disadvantage is the overhead associated with reading elements out of the data buffer and storing elements in the data buffer. Experience showed that the loss of speed is relatively small compared to the gain in structure.
The functions defined later will pull out all data they need to perform certain tasks and put in the results. Elements of `var` will be referenced by the point–operator: `var.x`. Note that this is **invalid** Gauss syntax.

D.2 Analyzing VAR Models

In addition to the vector autoregressive part the analysed VAR may contain a deterministic term, an exogenous term, and a term that contains a multiplicative combination of deterministic and endogenous or exogenous variables (mixed term). The variable types reflect the different parts of the underlying VAR model:

- Current endogenous variables (y_t)
- Lagged endogenous variables (y_{t-i})
- Exogenous variables (x_t)
- Deterministic variables (d_t)
- Mixed variables (z_t)

D.2.1 Reduced form model

The K dimensional VAR(p) model in reduced form is

$$y_t = AY_{t-1} + BX_t + Cd_t + FZ_t + u_t \tag{D.1}$$

where

y_t is the K dimensional endogenous variable,

$AY_{t-1} = A_1 y_{t-1} + \ldots + A_p y_{t-p}$ is the lagged endogenous term,

$BX_t = B_0 x_t + B_1 x_{t-1} + \ldots + B_q x_{t-q}$ is the exogenous term,

Cd_t is the deterministic term, and

FZ_t is the mixed term.

The following dimensions are assumed in the following

y_t	$(K, 1)$
A	(K, Kp)
x_t	$(L, 1)$
X_t	$(L(q+1), 1)$
B	$(K, L(q+1))$
d_t	$(M, 1)$
C	(K, M)
Z_t	$(N, 1)$
F_t	(K, N)

The procedure for defining a VAR(p) is `vml_InitVar()`. See the documentation and the examples at the end of this Chapter of how to use it.

D.2.2 Structural form model

The structural VAR model is

$$\widetilde{A}_0 y_t = \widetilde{A} Y_{t-1} + \widetilde{B} X_t + \widetilde{C} d_t + \widetilde{F} Z_t + u_t \tag{D.2}$$

where the setup is as in model (D.1) but with a parameter matrix $\widetilde{A}_0$ modelling the contemporaneous effects and structural form parameter matrices $\widetilde{A}$, $\widetilde{B}$, $\widetilde{C}$, and $\widetilde{F}$. Structural models are defined from reduced form models by adding a subset restriction matrix for $\widetilde{A}_0$. See Section D.2.4 for specifying subset restrictions.

D.2.3 Defining mixed variables

The library `var` allows to combine deterministic variables (v_t) and stochastic variables (w_t) multiplicatively:

$$z_t = v_t w_t.$$

It is required that w_t is contained in the endogenous or exogenous variables. The advantage over adding the variable z_t directly to the model is that the impulse response analysis and the bootstrap can process that information.
The variable z_t is coded using two matrices. The deterministic part is defined in a (T, M) matrix V. The rule is defined in a $(N, 4)$ matrix VW where the $Z_{(.,i)}$ is defined as

$VW_{(i,1)}$ specifies the variable type. The following variable types are valid:

	coding
endogenous	enl (level) end (differenced)
exogenous	ex
EC-term	ec
determin	de

$VW_{(i,2)}$ the (column) index of the variable,

$VW_{(i,3)}$ the lag, and

$VW_{(i,4)}$ the (column) index of the deterministic variable V, or $VW_{(i,4)} = 0$ if $z_t = w_t$.

Mixed variables are defined with the procedure call `vml_SetModelExtensions()`. This operation adds the elements `var.mx_c`, `var.Z` and optionally `var.mx_tf` to the `var` databuffer.

D.2.4 Linear restrictions

General linear restrictions:

Linear restrictions can be specified for each parameter matrix of a variable type in the form

$$g_i = \text{vec}(G_i) = R_i\gamma_i + r_i, \tag{D.3}$$

where G_i is the parameter matrix of the variable type i. The matrix G_i is thus one of the parameter matrices of model (D.1) or (D.2). It is not possible to set restrictions across different variable types.

Subset restrictions:

A subset restriction excludes an arbitrary variable from an equation. The associated parameter is set to zero. A subset restriction matrix has the same dimension as its associated parameter matrix. A zero element indicates that the respective parameter is set to zero. Subset restriction matrices can be specified for all parameter matrices of model D.1 or D.2. If a subset restriction matrix for $\widetilde{A}_0$ is specified the software assumes to deal with a structural model. Restrictions can be added with the `vml_SetRestrictions()` procedure to an existing VAR model. The table shows the keys the program uses to identify the different restriction matricies:

type	R_i	r_i	S_i
endogenous	R_A0	R_A0_C	S_A0
lagged endogenous	R_A	R_A_C	S_A
exogenous	R_B	R_B_C	S_B
deterministic	R_C	R_C_C	S_C
mixed	R_F	R_F_C	S_F

If for a coefficient matrix new linear restrictions are specified the old restrictions are replaced. General linear restrictions and subset restrictions are mutually exclusive. Specifying one restriction type for a coefficient matrix automtically removes the other restriction type if previously entered.

D.2.5 Estimation

The defined VAR model is estimated subject to the optional parameter constraints and optional estimation method. If no parameter constraints were specified a full VAR is estimated. If the etsimation procedure is not forced to apply a certain estimation method the estimation method is choosen according to the model set up. In this case the program selects on of the following methods:

OLS Ordinary least squares. Reduced form (full) VARs.

GLS Generalized least squares, or seemingly unrelated regression (SUR). Reduced form VARs with linear restrictions.

3SLS Three stages least squares. Structural VARs.

After estimation (see `vml_EstimateModel()` the data buffer defining the VAR model (`var`) contains the applied estimation method (`var.em_sys_x`), estimates for the model coefficients (`var.A0`, `var.A`, ...); their covariance ma-

trix (`var.cv_coeff`); estimates for the residuals (`var.u`), and for the residual covariance matrix (`var.cv_u`). In addition it is possible to compute standard error, t-values and p-values of the model coefficients by the procedure `vml_ComputeModelStatistics()`.

OLS Estimation

The ordinary least–squares (OLS) estimator is given by

$$\widehat{\beta}_{OLS} = (X'X)^{-1}X'Y.$$

The OLS estimator does not take into account the correlation structure of the disturbances accross equations. It neglegts information. It is therefore generally less efficient than the GLS estimator. However, there are two exceptions: The case of zero correlation between equations and the case of same regressors in each equation.
Although for efficiency reasons the OLS estimator should only be used on full VAR models it is provided here also for VAR models with parameter constraints.

GLS Estimation

The GLS estimator minimizes the generalized sum of squared errors $u\not{\Sigma}_u^{-1}u'$. Usually $\not{\Sigma}_u$ is unknown but can be estimated. The estimation is controlled by the global variable `__var_gls_cvu`:

`__var_gls_cvu = 0` $\not{\Sigma}_u$ estimated by unrestricted OLS (fastest), **default**,

`__var_gls_cvu = -1` $\not{\Sigma}_u$ estimated by restricted OLS,

`__var_gls_cvu = C` where C is a (K, K) matrix: $\not{\Sigma}_u = C$.

The residual covariance matrix estimator $\widehat{\not{\Sigma}}_u$ is

$$\widehat{\not{\Sigma}}_u = c^{-1}\sum_{i=1}^{T}\hat{u}_i * \hat{u}_i'$$

The divisor c is controlled by __var_div:

__var_div = 0 $c = T$, **default**,

__var_div = 1 $c = T - (M/K)$, where M is the number of estimated parameters in the restricted model and K is the number of equations.

GLS estimation can be performed itertively. The estimate for Σ_u in the i-th step is taken for parameter estimation in the $(i+1)$-th step. The number of iterations is controlled by two global variables __var_iter and __var_tol. __var_iter determines the maximum number of iterations. Default is 1 which means that GLS estimation is performed only once. __var_tol determines a convergence value for Σ_u. The estimation procedure stops if either the maximum number of iterations is reached or

$$\text{__var\_tol} \geq \left| \frac{\log(\det(\widehat{\Sigma}_u^{(i+1)})) - \log(\det(\widehat{\Sigma}_u^{(i)}))}{\log(\det(\widehat{\Sigma}_u^{(i)}))} \right|.$$

The default for the tolerance is __var_tol$= 10^{-4}$.

3SLS Estimation

It is well known (e.g. Dhrymes Ch.4) that the OLS estimator of structural parameters is biased and inconsistent. Consistent methods are the two-stage least squares estimator and the three-stage least square estimator.

D.2.6 Impulse response analysis

The impulse response analysis provides estimates for the forecast error and orthogonal impulse responses. Forecast error impulse responses are recursively computed as

$$\Phi_i = \sum_{j=1}^{i} \Phi_{i-j} A_j, \qquad i = 1, 2, \ldots$$

with $\Phi_0 = I_K$ and $A_j = 0$ for $j > p$. Orthogonal impulse responses take into account the correlation structure of the residuals. They are computed as

$$\Theta_i = P\Phi_i,$$

where $P'P = \Sigma_u$. One can see that $\Phi_i = \Theta_i$ if $\Sigma_u = I_K$. If a model is specified in structural form only forecast error impulse responses are computed.
In addition to the point estimates $\widehat{\Phi}_i$ and $\widehat{\Theta}_i$ the following bootstrap confidence intervals can be computed:

- Efron's percentile (CI_{EP})
- Hall's percentile (CI_{HP})
- Studentized (CI_{SH})
- Iterated (CI_{IT})

Bootstrap confidence intervals

Before computing bootstrap confidence intervals one should have gone through a sufficient model specification. Some violations of model assumtions can cause the algorithm to break down due to uninvertability of some moment matrices. Apart from econometric reasons one should avoid for the sake of computationality heavy model underfitting. Poor model specification can lead to explosive bootstrapped time series which in turn is the reason for the mentioned problems with matrix inversion. On the other hand model overfitting does not harm the algorithm but may lead to results that leave no space for qualitative interpretation.
Bootstrap methods are computer intensive methods. The computation time depends on the size of the VAR–model (number of parameters), the estimation method, the number of bootstrap drawings, and the bootstrap method. In general CI_{EP} and CI_{HP} take less time than CI_{SH} and CI_{IT}. All procedures have

a progress notification mechanism which informs about the approximate time to finish. The global variable __var_time is set to the minimum time between two progress notification messages (in seconds). The default is 5 (seconds). If the time to finish exceeds ones patience one can stop the computation, reparameterize the problem (usually take less bootstrap drawings) and restart the program.

The intervals CI_{EP} and CI_{HP} are computed in one procedure call. The iteration procedure iterates one of the mentioned intervals. This means that one can compute an iterated CI_{EP} and iterated CI_{HP}, an iterated CI_{SH}, and an iterated CI_{IT}.

Plotting impulse responses

There are various procedures for plotting impulse responses. The main plot procedure (and most complex in terms of input arguments) is vml__IRA_plotIR4CI4(). All other plot procedures have a reduced number of input arguments and call internally the main plot routine.

The user can control main title, plot color, line type, and labeling. A combination of impulse and response variables can also be selected.

D.3 Analyzing VECMs

The library var can analyse K dimensional VECMs of the very general form

$$\begin{aligned} \widetilde{\Gamma}_0 \Delta y_t &= \widetilde{\alpha}(\beta y_{t-1} + \beta^{(d)} d_t^{(ec)}) \\ &\quad + \widetilde{\Gamma} \Delta Y_{t-1} + \widetilde{B} X_t + \widetilde{C} d_t^{(sys)} + \widetilde{F} Z_t + u_t. \end{aligned} \tag{D.4}$$

It is possible to analyse reduced form VECMs ($\widetilde{\Gamma}_0 = I_K$) and structural VECMs. The cointegration relation ($ec_{t-1} = \beta y_{t-1} + \beta^{(d)} d_t^{(ec)}$) can be modelled with deterministic variables $d_t^{(ec)}$. Beside the lagged endogenous term

($\widetilde{\Gamma}\Delta Y_{t-1}$), the model may contain a term of current and lagged exogenous variables ($\widetilde{B}X_t$), of deterministic variables ($\widetilde{C}d_t^{(sys)}$), and time varying parameters that are partly summarized in the term $\widetilde{F}Z_t$.
Linear parameter constraints can be imposed on all parameter matrices of equation (D.4): $\widetilde{\Gamma}_0$, $\widetilde{\alpha}$, β, $\beta^{(d)}$, $\widetilde{\Gamma}$, $\widetilde{B}$, $\widetilde{C}$, and $\widetilde{F}$.

D.3.1 Reduced form model

The basic reduced form VECM is

$$\Delta y_t = \alpha\beta y_{t-1} + \Gamma\Delta Y_{t-1} + BX_t + Cd_t + u_t \tag{D.5}$$

where

$$\Gamma\Delta Y_{t-1} = \Gamma_1\Delta y_{t-1} + \ldots + \Gamma_{p-1}\Delta y_{t-p+1}$$

is the term of lagged endogenous variables,

$$BX_t = B_0x_t + B_1x_{t-1} + \ldots + B_{t-q}x_{t-q+1}$$

is the term of current and lagged exogenous variables and

$$Cd_t$$

is the term of deterministic variables.
Programatically equation (D.5) is set up by defining a VAR(p) model, imposing a cointegration rank r or setting r cointegration relations.

See `vml_InitVAR()`

`vml_SetCointRank()`

`vml_SetCointRelation()`

D.3.2 Structural VECM

The structural VECM is of the form

$$\widetilde{\Gamma}_0 \Delta y_t = \widetilde{\alpha}\beta y_{t-1} + \widetilde{\Gamma}\Delta Y_{t-1} + \widetilde{B}X_t + \widetilde{C}d_t + u_t. \tag{D.6}$$

A structural VECM is set up by defining a reduced form VECM and by imposing parameter constraints on the structural coefficient matrix $\widetilde{\Gamma}_0$.

See `vml_SetRestrictions()`

D.3.3 Modelling time dependent parameters

A time dependent parameter ξ (ξ can be any off–diagonal element of $\widetilde{\Gamma}_0$, or any element of $\widetilde{\alpha}$, $\widetilde{\Gamma}$, or $\widetilde{B}$) for the associated variable w_t (hence any element of Δy_t, ec_{t-1}, dY_{t-1}, X_t) is modelled as $\xi = \xi_1 + \xi_2 v_t$ where v_t is a deterministic variable:

$$\xi w_t = \xi_1 w_t + \xi_2 z_t \qquad \text{with} \quad z_t = v_t w_t$$

$\xi_1 w_t$ is already contained in D.6, v_t is added as a new explanatory variable to the system with parameter matrix ξ_2.

For the VECM model w_t can be $y_{i,t-j}$, $dy_{i,t-j}$, $ec_{i,t-1}$, or $x_{i,t-j}$. The indices i and j must run in the valid range. The z_t are collected in Z_t and are introduced with parameter matrix F to model (D.5) or (D.6).

With time dependent coefficients equation (D.6) extends to

$$\widetilde{\Gamma}_0 \Delta y_t = \widetilde{\alpha}\beta y_{t-1} + \widetilde{\Gamma}\Delta Y_{t-1} + \widetilde{B}X_t + \widetilde{C}d_t + \widetilde{F}Z_t + u_t.$$

The variable Z_t is coded with two matrices. The deterministic part is defined in a (T, M) matrix V. The variable type of w_t, the indices i and j, and the index of the deterministic variable v_t in V are defined in a $(N, 4)$ matrix VW:

$VW_{(l,1)}$ specifies the variable type of w_t The following variable types are valid:

w_t	variable	$VW_{(l,1)}$
y_{t-j}	endogenous (in levels)	enl
dy_{t-j}	endogenous (in differences)	end
x_{t-j}	exogenous	ex
ec_{t-1}	EC-term	ec
d_t	deterministic	de

$VW_{(l,2)}$ specifies the element in y_t, dy_t, x_t, or ec_{t-1};

$VW_{(l,3)}$ is the lag, j;

$VW_{(l,4)}$ specifies the (column) index of the deterministic variable V, or $VW_{(l,4)} = 0$ if $z_t = w_t$.

See `vml_SetModelExtensions()`

`vecm.mx_c`

`vecm.mx_tf`

`vecm.Z`

D.3.4 Restrictions

Cointegration rank

The cointegration rank r is either directly imposed to the model or indirectly imposed by adding r cointegration relations to the model.

If r is directly imposed to (D.4) it is required that $0 \leq r \leq K$. For $r = 0$ a VAR$(p-1)$ of the differenced y_t is estimated. For $r = K$ a VAR(p) in levels is estimated.

If r cointegration relations are set to (D.4) it is required that $0 < r < K$. In this case a (r, K) matrix β and (r, M_1) matrix $\beta^{(ec)}$ are specified and imposed

to the model. The cointegration rank results in this case from the number of cointegration relations.

See `vml_SetCointRank()`

`vml_SetCointRelation()`

`vml_AddCointRelation()`

`vecm.r`

`vecm.beta`

`vecm.beta_d`

Restrict deterministic variables to cointegration relation

By default the error correction term is modelled as

$$ec_t = \beta y_t$$

such that the VECM looks like

$$\Delta y_t = \alpha\beta y_{t-1} + \Gamma\Delta Y_{t-1} + BX_t + Cd_t + u_t$$

It is possible to constrain some (or all) variables of d_t to ec_t such that

$$ec_t = \beta y_t + \beta^{(d)} d_t^{(ec)}.$$

In this case the VECM becomes

$$\Delta y_t = \alpha(\beta y_{t-1} + \beta^{(d)} d_t^{(ec)}) + \Gamma\Delta Y_{t-1} + BX_t + Cd_t^{(sys)} + u_t$$

When restricting deterministic variables to ec_t the variables in d_t are split to $d_t^{(ec)}$ and $d_t^{(sys)}$. Note the index t of $d_t^{(ec)}$ although the endogenous variables y_{t-1} enter with time index $t-1$. For an arbitrary element $d_{i,t}$ of d_t one can write

$$\begin{aligned} d_{i,t} &= d_{i,t} - d_{i,t-1} + d_{i,t-1} \\ &\quad \Delta d_{i,t} + d_{i,t-1} \end{aligned}$$

and add $d_{i,t-1}$ to ec_{t-1} and leave $\Delta d_{i,t}$ at the i–th position of d_t. This makes sense if $d_{i,t}$ is a step dummy but makes problems if $d_{i,t}$ is a constant or trend variable. In order to circumvent the problems the above explained solution is used.

See `vml_RestrictDetsToCoint()`

`vecm.mx_d2ci`

Parameter constraints

Parameter constraints on (D.5) or (D.6) are imposed in the same manner as in Section D.2.4. Constraints on A_0 and A have no effect.

Parameter constraints are imposed by adding R_i and r_i, or S_i matrices to the already defined VECM. Restriction matrices are associated with the following keys to the respective parameter matrices:

i	R_i	r_i	S_i
Γ_0	`R_G0`	`RC_G0`	`S_G0`
Γ	`R_G`	`RC_G`	`S_G`
B	`R_B`	`RC_B`	`S_B`
C	`R_C`	`RC_C`	`S_C`
F	`R_F`	`RC_F`	`S_F`
α	`R_alpha`	`RC_alpha`	`S_alpha`

See `vml_SetRestrictions()`

`vecm.R_xxx`

`vecm.RC_xxx`

`vecm.S_xxx`

D.3.5 Estimation

The VECM can be estimated with the Johansen procedure or with a two stage procedure. Which estimation method is applied by the estimation procedure depends on the estimation method set before estimating the VECM. If no estimation method is set the two stage procedure is choosen automatically. If the Johansen procedure is desired one must set the estimation method to ``JOH''. If it is set to a method explained for the VAR case this method is applied in the second stage (by OLS, GLS, 3SLS).

For the two stage procedure one can also define the method for estimating the cointegration relation. Possible methods are the Johansen procedure and a single equation error correction regression. The first stage runs a regression for the cointegration relation, the second step plugs in the results of the first regression and estimates the system. How the cointegration relation is estimated is controlled with the data buffer element `vecm.em_cr`. If `vecm.em_cr = ``JOH''` the cointegration vectors are estimated by the Johansen procedure. If `vecm.em_cr = ``s&w''` the cointegration vectors are estimated by a single equation error correction model. In all other cases it is assumed that `vecm` already knows β and $\beta^{(d)}$.

See `vml_SetEstimationMethod()`

`vml_SetEstimationMethodCIR()`

`vml_EstimateModel()`

`vecm.em_sys`

`vecm.em_cir`

Johansen procedure

The Johansen procedure is implemented in this library such that no structural VECM is valid, exogenous variables are skipped, and no parameter constraints

are regarded. The only restriction regarded are the cointegration rank and the deterministic variables that are restricted to the cointegration relation.

Two stage procedure

The two stage procedure determines in the first step the cointegration relation and estimates the VECM in the second step.

Specifying the cointegration vector from extern source:
The knowledge of the cointegration relation may come from economic theory, or some extern pre–analysis. Therefore one can add cointegration relations directly to the already defined VECM model. The general form of the error correction term is given in (??). Setting or adding β and $\beta^{(d)}$ to a already defined VECM implies that the error correction variable ec_t is recomputed and if necessary the mixed variables involving ec_t are updated (in Z). It is also possible to estimate r_1 cointegration relations from the data set at hand and to set r_2 cointegration relations by hand. In this case always the first r_1 cointegration vectors are set by the software and the next r_2 cointegration vectors stay unchanged.

See `vml_SetCointRelation()`

`vml_AddCointRelation()`

`vecm.beta`

`vecm.beta_d`

Estimating the cointegration vector from the data:
Estimation of the cointegration vector(s) can be done with a single equation error correction regression or with the Johansen approach. For that one has to set explicitly the method for estimating the cointegration relation either to "JOH" for the Johansen approach or to "s&w" for the single equation error

correction regression. If no method is set it is assumed that the cointegration vectors were already specified. Estimating the cointegration relation with the Johansen approach is subject to the restrictions lined out in the paragraph describing this implementation. It still might be advantageous since one can also estimate more than one cointegration relation. The single equation error correction regression only allows to estimate only one cointegration relation. Here one effectively estimates one equation of (D.5). The first equation is taken by default. All parameter constraints are adopted to the single equation regression. The resulting estimates $\widehat{\beta}$ and $\widehat{\beta}^{(d)}$ are normed by the first element of $\widehat{\beta}$. It is also possible to specify a different equation and a different norming coefficient. It is also possible to define a completely new problem for the single equation regression with its own set of exogenous and deterministic variables or with completely different parameter constraints.

See `vml_SetEstimationMethodCIR()`

`vml_SetEquationIndexCIR()`

`vml_SetNormingIndexCIR()`

`vml_InitCIR()`

`vml_SetRestrictionCIR()`

`vml_SetCIRModel()`

`vml_EstimateCIR()`

`vecm.em_cir`

`vecm.beta`

`vecm.beta_d`

`vecm.cir`

`vecm.cirEqIdx`

`vecm.cirNoIdx`

Estimating the System:

The system can be estimated with OLS, GLS, or 3SLS.

VAR representation

After estimating the VECM it is automatically rewritten as a VAR. Since this is a algebraic reformulation of the terms that include endogenous variables the exogenous variables and deterministic variables are omitted in the next lines:

$$\begin{aligned}
\widetilde{\Gamma}_0 \Delta y_t &= \widetilde{\Pi} y_{t-1} + \widetilde{\Gamma} \Delta Y_{t-1} + u_t \\
\widetilde{\Gamma}_0 (y_t - y_{t-1}) &= \widetilde{\Pi} y_{t-1} + \widetilde{\Gamma}_1 (y_{t-1} - y_{t-2}) + \ldots + \widetilde{\Gamma}_{p-1} (y_{t-p+1} - y_{t-p}) + u_t \\
\widetilde{\Gamma}_0 y_t &= (\widetilde{\Pi} + \widetilde{\Gamma}_0 + \widetilde{\Gamma}_1) y_{t-1} + (\widetilde{\Gamma}_1 - \widetilde{\Gamma}_2) y_{t-2} + \ldots + (-\widetilde{\Gamma}_{p-1}) y_{t-p} + u_t \\
\widetilde{A}_0 y_t &= \widetilde{A}_1 y_{t-1} + \widetilde{A}_2 y_{t-2} + \ldots + \widetilde{A}_p y_{t-p} + u_t
\end{aligned}$$

with $\widetilde{\Pi} = \widetilde{\alpha}\beta$, $\widetilde{A}_0 = \widetilde{\Gamma}_0$, $\widetilde{A}_1 = \widetilde{\Pi} + \widetilde{\Gamma}_0 + \widetilde{\Gamma}_1$, $\widetilde{A}_i = \widetilde{\Gamma}_i - \widetilde{\Gamma}_{i-1}$ $(i = 2, \ldots, p-1)$, and $\widetilde{A}_p = -\widetilde{\Gamma}_{p-1}$.

Special attention is given the mixed variables that include (differenced) endogenous variables and error correction variables. Depending on the form of ec_t this can have an effect on the deterministic variables and the mixed variables itself.

See `vecm.mx_c_ec`

`vecm.mx_c_ar`

D.3.6 Screen output

The screen output procedures print the input data and estimation results in a clear way to the output screen.

See `vml_ShowInput()`

`vml_ShowEstimationResults()`

D.3.7 Impulse response analysis

Since model (D.5) has a VAR–representation which is also returned by the the estimation procedure one can right away apply the tools for the impulse response analysis introduced in Section D.2.6.

For the bootstrap methods one should be aware that the bootstrap time series y_t^* also updates ec_t. If a two stage procedure is choosen the cointegration relation is reestimated in every bootstrap drawing if the estimation method for the cointegration relation is not set to `''none''`

D.3.8 Data buffer elements in the VECM analysis

variable	dimension	explanation
`vecm.r`	$(1, 1)$	cointegration rank r
`vecm.ec`	$(p + T, r)$	error correction term $(ec_{-p+1}, \ldots, ec_T)'$

D.3.9 GAUSS procedures

commutationMatrix

Purpose

Generates commutation matrix for a (M, N)-matrix.

Format

```
y = commutationMatrix(M, N);
```

Input

`M (1,1)` Number of rows

`N (1,1)` Number of columns

Output

`y (M*N,M*N)` Commutation matrix for any (M, N)-matrix

Remarks

See Lütkepohl (1993a) Chapter A.12.2 for the definition of this special purpose matrix.

Source

`var_tool.src`

duplicationMatrix

Purpose

Generates duplication matrix for a K-dimensional symmetric matrix.

Format

```
y = duplicationMatrix(K);
```

Input

`K (1,1)` Dimension of symmetric matrix

Output

`y (K*K,0.5*K*(K+1))` Duplication matrix for vectorized lower triangular symmetric matrix

Remarks

See Lütkepohl (1993a) Chapter A.12.2 for the definition and usage of this special matrix.

Source

`var_tool.src`

eliminationMatrix

Purpose

Generates elemination matrix for a K-dimensional square matrix.

Format

```
y = eliminationMatrix(K);
```

Input

`K (1,1)` Dimension of square matrix

Output

`y (0.5*K*(K+1),K*K)` Elemination matrix for vectorized K-dimensional square matrix

Remarks

See Lütkepohl (1993a) Chapter A.12.2 for the definition of this special matrix.

Source

`var_tool.src`

smakewin

Purpose

Configures plot window such that a main title appears.

Format

```
smakewin(numrows, numcols, shift, wintype, );
```

Remarks

Procedure developed by Sune Karlsson.

Source

var_irap.src

smwintit

Purpose

Configures plot window such that a main title appears.

Format

```
titwin = SMWINTIT(ROWS, COLS, TITLE, TYPE, );
```

Remarks

Procedure developed by Sune Karlsson.

Source

var_irap.src

var__EstimateVECModel

Purpose
Estimates VECM.

Format
```
modelHat = var__EstimateVECModel(model);
```

Input

`model (Y,1) data buffer` Data buffer: defined VEC.

Output

`modelHat (Z,1) data buffer` Estimated VEC model as data buffer.

Remarks
This procedure simply estimates the data buffer `model` and returns and it with the estimated parameter matrices.
Changes in data buffer: `+ beta, alpha, G0, G, A0, A, B, C, F`

See
var_EstimateModel

Source
`var_est2.src`

var__IRA

Purpose

Computes impulse responses.

Format

{feir, orir} = var__IRA(model, irmax);

Input

model	(N,1) data buffer	Estimated VAR model.
irmax	(1,1) matrix	Time horizon for computing impulse responses.

Output

feir	(1+ir_max,K*K) matrix	Forecast error impulse responses
orir	(1+ir_max,K*K) matrix	Orthogonalized impulse responses
	(0,0) matrix	If model is in structural form.

Remarks

Impulse responses are computed for the model

$$A_0 y_t = A_1 y_{t-1} + \ldots + A_p y_{t-p}$$

with $A = [A_1 : A_2 : \ldots : A_p]$

This procedure distinguishes between reduced form models and structural form models. Reduced form models are identified if the argument $A_0 = I_K$, or $A_0 = \{\}$.

Impulse responses for reduced form models are computed by

$$\Phi_i = \sum_{j=1}^{i} \Phi_{i-j} A_j, \qquad i = 1, 2, \ldots$$

(see Lütkepohl (1993a)) with $\Phi_0 = I_K$. When impulse response for structural form models are computed the coefficient matrices are first transformed to reduced form coefficient matrices and then the recursion above is used using $\Phi_0 = A_0^{-1}$. This procedure fails if A_0 is not invertible. If the matrix A_0 is specified no orthogonal impulse responses are computed.

The impulse response functions are ordered row-wise, i.e. the first K columns in row $i+1$ in the output argument feir take the quantities of the first row of Φ_i and so forth.

Source

var_ira.src

var__IRA_CI_HEP

Purpose

Computes Hall and Efron percentile bootstrap confidence intervals for impulse responses of a given VAR-model.

Format

```
{irf_ferr_hat, ferr_ep_lower, ferr_ep_upper, ferr_hp_lower,
ferr_hp_upper, irf_orth_hat, orth_ep_lower, orth_ep_upper,
orth_hp_lower, orth_hp_upper, newSeed} =
   var__IRA_CI_HEP(var, nob, seed, ir_max, coverage);
```

Input

var	(N,1) matrix	Data buffer with model information
nob	(1,1) matrix	Number of bootstrap replications
seed	(1,1) matrix	Seed for drawing residuals (seed=0 for using no seed)
ir_max	(1,1) matrix	Maximum number impulse responses to compute
coverage	(1,1) matrix	Coverage of confidence intervals, in (0,1)

Output

irf_ferr_hat	(1+ir_max,K*K) matrix	Point estimates forecast error impulse responses
ferr_ep_lower	(1+ir_max,K*K) matrix	Lower bound for percentile confidence interval (Efron) of forecast error impulse responses
ferr_ep_upper	(1+ir_max,K*K) matrix	Upper bound for percentile confidence interval (Efron) of forecast error impulse responses
ferr_hp_lower	(1+ir_max,K*K) matrix	Lower bound for percentile confidence interval (Hall) of forecast error impulse responses
ferr_hp_upper	(1+ir_max,K*K) matrix	Upper bound for percentile confidence interval (Hall) of forecast error impulse responses
irf_orth_hat	(1+ir_max,K*K) matrix	Point estimates orthogonal impulse responses

`orth_ep_lower`	`(1+ir_max,K*K)`	`matrix`	Lower bound for percentile confidence interval (Efron) of orthogonal impulse responses
`orth_ep_upper`	`(1+ir_max,K*K)`	`matrix`	Upper bound for percentile confidence interval (Efron) of orthogonal impulse responses
`orth_hp_lower`	`(1+ir_max,K*K)`	`matrix`	Lower bound for percentile confidence interval (Hall) of orthogonal impulse responses
`orth_hp_upper`	`(1+ir_max,K*K)`	`matrix`	Upper bound for percentile confidence interval (Hall) of orthogonal impulse responses
`newSeed`	`(1,1)`	`matrix`	updated seed

Remarks

Computes Hall and Efron percentile bootstrap confidence intervals. Assumes that model is already estimated and in VAR-form.

The impulse response functions (and the confidence bounds matrices) are ordered row-wise, i.e. the first K columns in row $i+1$ in the output argument `irf_ferr_hat` take the quantities of the first row of Φ_i and so forth.

Globals

__var_Note_StartTime

Source

`var_irc1.src`

var__IRA_CI_IT

Purpose

Computes iterated bootstrap confidence intervals for impulse responses of a given VAR-model.

Format

```
{irfeHat, irfe_ci1_lo, irfe_ci1_up, irfe_ci2_lo, irfe_ci2_up,
irorHat, iror_ci1_lo, iror_ci1_up, iror_ci2_lo, iror_ci2_up,
seed, seed_sd, seed_it} =
   var__IRA_CI_IT(var, nob, nob_sd, nob_it, seed, seed_sd,
                  seed_it, ir_max, coverage, ciType,
                  tuneType);
```

Input

var	(N,1)	matrix	Data buffer with model information
nob	(1,1)	matrix	Number of bootstrap replications
nob_sd	(1,1)	matrix	Number of bootstrap replications when estimating the standard deviation.
nob_it	(1,1)	matrix	Number of bootstrap replications in the iteration stage.
seed	(1,1)	matrix	Seed for drawing residuals. (seed=0 for using no seed)
seed_sd	(1,1)	matrix	Seed for drawing residuals when estimating the standard deviation. (seed=0 for using no seed)
seed_it	(1,1)	matrix	Seed for drawing residuals in the iteration stage. (seed=0 for using no seed)
ir_max	(1,1)	matrix	maximum number impulse responses to compute
coverage	(1,1)	matrix	Coverage of confidence intervals, in (0,1)
ciType	(1,1)	proc	Procedure pointer, defines procedure for computing confidence intervals. Valid procedures are var__IRA_CI_HEP and var__IRA_CI_ST.
tuneType	(1,1)	proc	Procedure pointer, for computing tuning parameter. only var__IRA_CI_IT_tune1 is allowed at the moment.

Output

irf_ferr_hat	(1+ir_max,K*K)	matrix	Point estimates forecast error impulse responses

ferr_ep_lower	(1+ir_max,K*K)	matrix	Lower bound for iterated percentile confidence interval (Efron) of forecast error impulse responses
ferr_ep_upper	(1+ir_max,K*K)	matrix	Upper bound for iterated percentile confidence interval (Efron) of forecast error impulse responses
ferr_hp_lower	(1+ir_max,K*K)	matrix	Lower bound for iterated percentile confidence interval (Hall) of forecast error impulse responses
ferr_hp_upper	(1+ir_max,K*K)	matrix	Upper bound for iterated percentile confidence interval (Hall) of forecast error impulse responses
irf_orth_hat	(1+ir_max,K*K)	matrix	Point estimates orthogonal impulse responses
orth_ep_lower	(1+ir_max,K*K)	matrix	Lower bound for iterated percentile confidence interval (Efron) of orthogonal impulse responses
orth_ep_upper	(1+ir_max,K*K)	matrix	Upper bound for iterated percentile confidence interval (Efron) of orthogonal impulse responses
orth_hp_lower	(1+ir_max,K*K)	matrix	Lower bound for iterated percentile confidence interval (Hall) of orthogonal impulse responses
orth_hp_upper	(1+ir_max,K*K)	matrix	Upper bound for iterated percentile confidence interval (Hall) of orthogonal impulse responses
seed	(1,1)	matrix	updated seed
seed_sd	(1,1)	matrix	updated seed
seed_it	(1,1)	matrix	updated seed

Remarks

Computes Hall and Efron iterated percentile bootstrap confidence intervals. Assumes that model is already estimated and in VAR-form.

The impulse response functions (and the confidence bounds matrices)

are ordered row-wise, i.e. the first K columns in row $i+1$ in the output argument `irf_ferr_hat` take the quantities of the first row of Φ_i and so forth.

Globals

__var_Note_StartTime

See

__var_time
var__IRA_CI_IT_tune1
var__IRA_CI_HEP
var__IRA_CI_ST

Source

`var_irc3.src`

var__IRA_CI_IT_in

Purpose
Helper method for iteration procedure.

Format
```
y = var__IRA_CI_IT_in(ci, value);
```

Input

ci	(N x 2)	matrix	Set of confidence intervals
value	(1 x 1)	matrix	Parameter that should be covered by ci

Output

y	(3 x 1) matrix	Absolute counts that value is in one of the three areas that are defined by ci

Remarks
This procedure returns the frequency that the number value is (a) below lower bound, (b) between lower and upper bound, or (c) above upper bound. The column sum of y is N, the number of rows of ci.

Globals
__var_Note_StartTime

See
var__IRA_CI_IT

Source
var_irc3.src

var__IRA_CI_IT_tune1

Purpose

Find a tuning parameter for the iterated bootstrap that adds to the confidence interval endpoints with the same length.

Format

```
t = var__IRA_CI_IT_tune1(Lc, theta, alpha);
```

Input

Lc	(N x 2)	matrix	set of confidence intervals
theta	(1 x 1)	matrix	true parameter
alpha	(1 x 1)	matrix	nominal coverage

Output

t	(1 x 1)	matrix	tuning parameter

Remarks

The returned tuning parameter is computed such that if it is added to both sides of the input confidence intervals, the real coverage is exactly the nominal coverage.

Globals

__var_Note_StartTime

See

var__IRA_CI_IT

Source

var_irc3.src

var__IRA_CI_ST

Purpose

Computes studentized bootstrap confidence intervals for impulse responses of a given VAR-model.

Format

```
{irf_ferr_hat, ferr_st_lower, ferr_st_upper, irf_orth_hat,
orth_st_lower, orth_st_upper, seed, seed_sd} =
   var__IRA_CI_ST(var, nob, nob_sd, seed, seed_sd, ir_max,
                  coverage);
```

Input

var	(N,1) matrix	Data buffer with model information
nob	(1,1) matrix	Number of bootstrap replications
nob_sd	(1,1) matrix	Number of bootstrap replications for estimating the standard deviation
seed	(1,1) matrix	Seed for drawing residuals (seed=0 for using no seed)
seed_sd	(1,1) matrix	Seed for drawing residuals when computing standard deviation (seed=0 for using no seed)
ir_max	(1,1) matrix	Maximum number impulse responses to compute
coverage	(1,1) matrix	Coverage of confidence intervals, in (0,1)

Output

irf_ferr_hat	(1+ir_max,K*K) matrix	Point estimates of forecast error impulse responses
ferr_st_lower	(1+ir_max,K*K) matrix	Lower bounds for studentized confidence interval of forecast error impulse responses
ferr_st_upper	(1+ir_max,K*K) matrix	Upper bounds for studentized confidence interval of forecast error impulse responses
irf_orth_hat	(1+ir_max,K*K) matrix	Point estimates of orthogonal impulse responses
orth_st_lower	(1+ir_max,K*K) matrix	Lower bounds for studentized confidence interval of orthogonal impulse responses

orth_st_upper	(1+ir_max,K*K)	matrix	Upper bounds for studentized confidence interval of orthogonal impulse responses
newSeed	(1,1)	matrix	updated seed
newSeed_sd	(1,1)	matrix	updated seed

Remarks

The impulse response functions (and the confidence bounds matrices) are ordered row-wise, i.e. the first K columns in row $i+1$ in the output argument `irf_ferr_hat` take the quantities of the first row of Φ_i and so forth.

Globals

__var_Note_StartTime

See

__var_Note_StartTime
vml_InitVAR

Source

var_irc2.src

var__johansenApproach_CPLX

Purpose

Checks for complex results.

Format

```
y = var__johansenApproach_CPLX(x, name);
```

Input

x	(N,M)	Some matrix, estimation result.
name	(1,1)	Name string

Output

y	(N,M)	The x-matrix, without imaginary term if neglectible.

Remarks

Procedure checks for complex input matrices. Depending on the algorithm the Johansen procedure might result with complex numbers with no or neglectible term. Used by `vml__johansenApproach`

Globals

__var_Note_StartTime

See

vml__johansenApproach

Source

var_joh.src

var__MakeHeadAndWindows

Purpose

Tiles graphic window in head window and a matrix of sub windows.

Format

```
shift = var__MakeHeadAndWindows(nrows, ncols, tit);
```

Input

`nrows (1,1) matrix`	Number of rows for sub window matrix.
`ncols (1,1) matrix`	Number of columns for sub window matrix.
`tit (1,1) string`	Main title.

Remarks

This procedure observes `_pdate` and displays the date in the head window. This procedure is a modified `smwintit()` proc by Sune Karlsson.

Globals

__var_Note_StartTime

See

smwintit

Source

var_irap.src

var__numberOfEquations

Purpose
Helper method for estimation procedures.

Format
```
noEq = var__numberOfEquations(model);
```

Input

`model (M,1) data buffer` Defined VAR or VEC model

Output

`noEq (1,1) matrix` number of equations.

Globals
__var_Note_StartTime

See
vml_InitVAR

Source
var_ls.src

var__resetPlotControls

Purpose

Resets global graphics controls from this library to default values.

Format

```
call var__resetPlotControls();
```

Globals

__var_Note_StartTime

See

__var_show_legend
__var_show_main_title
__var_show_plot_title
__var_show_date
__var_height_legend
__var_height_axis
__var_height_numbers
__var_height_plot_title
__var_height_main_title
__var_line_type
__var_line_color
__var_line_width
__var_legend_x
__var_legend_y
__var_legend_strings

Source

var_irap.src

var__showCoefficients

Purpose

Helper Method for some other show-methods.

Format

```
call var__showCoefficients(c, sd, tv, pv, sr, Nv, Neq,
                           label);
```

Input

c	(K,L*px)	Coefficients
	(0,0)	if empty no screen output
sd	(K,L*px)	Standard deviation of coefficients
tv	(K,L*px)	t-Statistik for coefficients
pv	(K,L*px)	p-values for coefficients
sr	(K,L*px)	Matrix of subset restrictions (matrix of zeros and ones where a zero indicates that the respective variable was excluded from the model)
Nx	(L,1)	Variable names
Neq	(K,1)	Equation names
label	(1,1)	Caption

Output

```
Screen output
```

Remarks

The procedure returns without screen output if x={}.

Globals

__var_Note_StartTime

See

__var_showPrecision
__var_showWidth
var__showVAR_Coefficients
var__showVEC_Coefficients

Source

var_show.src

var__showEC_Coefficients

Purpose

Prints estimation results of error correction term to the screen.

Format

```
call var__showEC_Coefficients(model);
```

Input

`model (M,1) data buffer` Estimated VEC model

Output

`Screen output`

Globals

__var_Note_StartTime

Source

var_show.src

var__showLegend

Purpose
Prints legend to the screen.

Format
```
call var__showLegend();
```

Input

`eqNames (M,1) string array` Names of equations or variables

Output

`Screen output`

Remarks
Mainly by other procedures used.

Globals
__var_Note_StartTime

See
var__showVAR_Coefficients
var__showVEC_Coefficients
var__showEC_Coefficients

Source
`var_show.src`

var__showVAR_Coefficients

Purpose
Prints VAR representation to the screen.

Format
```
call var__showVAR_Coefficients(model);
```

Input

`model (M,1) data buffer` Estimated VEC or VAR model

Output

`Screen output`

Globals
__var_Note_StartTime

Source
var_show.src

var__showVEC_Coefficients

Purpose
Prints VEC representation to the screen.

Format
```
call var__showVEC_Coefficients(model);
```

Input

`model (M,1) data buffer` Estimated VEC model

Output

`Screen output`

Globals
__var_Note_StartTime

Source
var_show.src

var__specifyRSSdivisor

Purpose

Automates the rule for choosing the devisor of the residual sum of squares when computing residual covariance matrix.

Format

```
var__specifyRSSdivisor(model);
```

Input

`model (M,1) data buffer` Defined VAR or VEC model

Output

`none.`

Remarks

Manipulates the global `__var_div`.

Globals

__var_Note_StartTime
__var_div

Source

`var_ls.src`

var__splitDeterministics

Purpose

Split deterministic observations to a part entering the cointegration relation and a part entering the VEC-System.

Format

{d_vec, d_ec} = var__splitDeterministics(d, idx_c, idx_l);

Input

d	(T,M) matrix	Deterministic variables, can be empty
idx_c	(N,1) matrix	Indizes of deterministic variables that enter the cointegration relation unlagged (with current values), can be empty
idx_l	(O,1) matrix	Indizes of deterministic variables that enter the cointegration relation with lagged values: The current difference is kept in the system block of deterministic variables, can be empty

Output

d_vec	(T,M-N) matrix	Deterministic variables, that enter system estimation
d_ec	(T,N+O) matrix	Deterministic variables restricted to cointegration relation

Remarks

idx_c and idx_l must be mutually exclusive!

The variables referenced by idx_l are written as

$$y_t = y_t - y_{t-1} + y_{t-1} = \Delta y_t + y_{t-1}$$

where Δy_t is kept in the block of deterministic variables for system estimation and y_{t-1} goes to the block of deterministic variables for estimating the cointegration relation.

Globals

__var_Note_StartTime

__var_div

See

var__splitDeterministicsNames

Source

var_ls.src

var__splitDeterministicsNames

Purpose

Creates names for deterministic variables according to the restriction rule.

Format

```
{Nd_vec, Nd_ec} =
   var__splitDeterministicsNames(Nd, idx_c, idx_l);
```

Input

Nd	(M,1) character matrix or string array	Names for deterministic variables, can be empty
idx_c	(N,1) matrix	Indizes of deterministic variables that enter the cointegration relation unlagged (with current values), can be empty
idx_l	(O,1) matrix	Indizes of deterministic variables that enter the cointegration relation with lagged values: The current difference is kept in the system block of deterministic variables, can be empty

Output

Nd_vec	(M-N,1) string array	Names of deterministic variables that enter system estimation
Nd_ec	(N+O,1) string array	Names of deterministic variables restricted to cointegration relation

Remarks

idx_c and idx_l must be mutually exclusive!

Globals

__var_Note_StartTime
__var_div

See

var__splitDeterministics

Source

var_ls.src

var_ComputeModelStatistics

Purpose

Computes various statistics from the estimated model.

Format

```
y = var_ComputeModelStatistics(model);
```

Input

`model (Y,1) data buffer` Estimated VAR or VEC.

Output

`y (Z,1) data buffer` Estimated VAR or VEC with additional statistics.

Remarks

This procedure should be applied after calling `vml_EstimateModel()`
Computes

- Matrix of standard deviation (`sd_...`),
- Matrix of t-values (`tv_...`),
- Matrix of p-values (`pv_...`).

Changes in data buffer: `sd_XXX, tv_XXX, pv_XXX`

Globals

__var_Note_StartTime
__var_div

See

vml_InitVAR
var_EstimateModel

Source

`var_est.src`

var_eigenvalues

Purpose

Returns the modulus of the eigenvalues of the characteristic polynomial.

Format

```
Eig = var_eigenvalues(Ay);
```

Input

`Ay (K,K*p)` Coefficient matrix

Output

`Eig (K*p,1)` Modulus of the eigenvalues of the characteristic polynomial of Ay

Globals

__var_Note_StartTime
__var_div

Source

var_var.src

var_EstimateECModel

Purpose

Estimates Error Correction Model.

Format

```
modelHat = var_EstimateECModel(model);
```

Input

`model (Y,1) data buffer`	Data buffer defined a VAR or VEC.

Output

`modelHat (Z,1) data buffer`	Estimated equation from VAR model that holds the error correction term.

Remarks

This procedure estimates a cointegration relation from a previously defined VAR or VEC model. The equation from which the error correction relation is derived must be defined before.

Changes in data buffer: `+ beta, alpha, G0, G, A0, A, B, C, F`

Globals

__var_Note_StartTime
__var_div

See

var_initCIR

Source

`var_est3.src`

var_EstimateModel

Purpose

Estimates VAR or VEC model.

Format

```
y = var_EstimateModel(model);
```

Input

`model (Y,1) data buffer` Data buffer defining a VAR or VEC.

Output

`y (Z,1) data buffer` Estimated VAR or VEC as data buffer.

Remarks

This procedure estimates the data buffer `model` and returns it with the estimated parameter matrices.

The procedure checks first if a cointegration rank r is defined. If $r = 0$ Δy_t has a stable VAR($p-1$) representation. For $r = K$ y_t is a stable VAR(p). This is considered in the estimation procedure.

Changes in data buffer: `beta, alpha, G0, G, A0, A, B, C, F`

Globals

__var_Note_StartTime
__var_div

See

vml_InitVAR
var_EstimateModel

Source

`var_est.src`

var_impulseResponses

Purpose

Computes impulse responses.

Format

```
{feir, orir} =
   var_impulseResponses(A0, A, cvu, ir_max, outFmt);
```

Input

A0	(K,K)	matrix	Parameter matrix of endogenous variables
	(0,0)		A0=
A	(K,p*K)	matrix	Parameter matrix of p lagged endogenous variables
cvu	(K,K)	matrix	Residual covariance matrix
ir_max	(1,1)	matrix	Time horizon for computing impulse responses.
outFmt	(1,1)	matrix	outFmt=1: output as (1+ir_max,K*K)-matrices
			outFmt=0: output as (1,K*K*(1+ir_max))-matrices

Output

feir	(1+ir_max,K*K)	matrix	Forecast error impulse responses, if outFmt=1
	(1,K*K*(1+ir_max))	matrix	if outFmt=0
orir	(1+ir_max,K*K)	matrix	Orthogonalized impulse responses, if outFmt=1
	(1,K*K*(1+ir_max))	matrix	if outFmt=0
	(0,0)	matrix	If $A0 \neq I_K$

Remarks

Impulse responses are computed for the model

$$A_0 y_t = A_1 y_{t-1} + \ldots + A_p y_{t-p}$$

This procedure distinguishes between reduced form models and structural form models. Reduced form models are identified if the argument $A_0 = I_K$, or $A_0 = \{\}$.

Impulse responses for reduced form models are computed by

$$\Phi_i = \sum_{j=1}^{i} \Phi_{i-j} A_j, \qquad i = 1, 2, \ldots$$

(see Lütkepohl (1993a)) with $\Phi_0 = I_K$. When impulse response for structural form models are computed the coefficient matrices are first transformed to reduced form coefficient matrices and then the recursion above is used using $\Phi_0 = A_0^{-1}$. This procedure fails if A_0 is not invertible. If the matrix A_0 is specified no orthogonal impulse responses are computed.

The impulse response functions are ordered row-wise, i.e. the first K columns in row $i + 1$ in the output argument `feir` take the quantities of the first row of Φ_i and so forth, if `outFmt=1`. In case of `outFmt=0` the just described output is vectorized and transposed.

Globals

__var_Note_StartTime
__var_div

Source

`var_ira.src`

var_indcv

Purpose

Checks one character vector against another and returns the indices of the elements of the first vector in the second vector.

Format

```
z = var_indcv(what, where);
```

Input

`what`	`(N,1) character matrix`	Contains the elements to be found in vector `where`
`where`	`(M,1) character matrix`	Vector to be searched for matches to the elements of `what`

Output

`z`	`(N,1) matrix`	Vector of integers containing the indices of the corresponding element of `what` in `where`

Remarks

This function is comparable to the Gauss function `indcv` but fills the elements of `what` with blanks.

Globals

__var_Note_StartTime
__var_div

Source

var_tool.src

var_initCIR

Purpose

Defines a data buffer for estimating the cointegration relation from one equation of a VAR-model.

Format

```
var =
   var_InitCIR(y, py, Ny, x, px, Nx, d, Nd, idx_equa,
               idx_norm);
```

Input

`y`	`(py+T,K)`	`matrix`	Time series of endogenous variables
`py`	`(1,1)`	`matrix`	Number of lagged endogenous variables ($:= p$)
`Ny`	`(K,1)`	`string array`	Names of endogenous variables
	`(0,0)`		`Ny={}`
`x`	`(px+T,L)`	`matrix`	Time series of exogenous variables
	`(0,0)`		`x={}` if no exogenous variables are defined in the system
`px`	`(1,1)`	`matrix`	Number of lagged exogenous variables (ignored if `x={}`) ($:= q$)
`Nx`	`(L,1)`	`string array`	Names of exogenous variables
	`(0,0)`		`Nx={}`
`d`	`(py+T,M)`	`matrix`	Time series of deterministic variables
	`(0,0)`		`d={}` if no deterministic variables are defined in the system
`Nd`	`(M,1)`	`string array`	Names of deterministic variables
	`(0,0)`		`Nd={}`
`idx_equa`	`(1,1)`	`matrix`	Index of equation that is used for estimating the cointegration relation.
`idx_norm`	`(1,1)`	`matrix`	Index of normalizing variable, i.e. the coefficient of this variable in the cointegration relation in 1.

Output

`var (N,1) data buffer`	Model for estimating the cointegration relation.

Remarks

This procedure generates the following fields [optional fields] in the returned data buffer: `y, py, idx_equa, idx_norm [,Ny, x, px, Nx, d, Nd]`

Globals

__var_Note_StartTime
__var_div

See

vml_InitVAR

Source

`var_init.src`

var_plotIR

Purpose
Plots one impulse response function.

Format
```
var_plotIR(irf);
```

Input

`irf (R,K*K)` Impulse response series.

Output

`Graphics`

Globals
__var_Note_StartTime
__var_div

See
var_plotIR4CI4

Source
`var_irap.src`

var_plotIR4CI4

Purpose

Generates plot of up to four impulse response functions and their confidence intervals.

Format

```
var_plotIR4CI4(irf1, irf2, irf3, irf4, ci1_l, ci1_u,
               ci2_l, ci2_u, ci3_l, ci3_u, ci4_l, ci4_u,
               mainTitle, impulseNames, responseNames,
               row_sel, col_sel);
```

Input

irf1	(R,K*K)	impulse response series 1
irf2	(R,K*K)	impulse response series 2
or	(0,0)	={} for none.
irf3	(R,K*K)	impulse response series 3
or	(0,0)	={} for none.
irf4	(R,K*K)	impulse response series 4
or	(0,0)	={} for none.
ci1_l	(R,K*K)	lower bound for impulse response series 1
or	(0,0)	={} for none.
ci1_u	(R,K*K)	upper bound for impulse response series 1
or	(0,0)	={} for none.
ci2_l	(R,K*K)	lower bound for impulse response series 2
or	(0,0)	={} for none.
ci2_u	(R,K*K)	upper bound for impulse response series 2
or	(0,0)	={} for none.
ci3_l	(R,K*K)	lower bound for impulse response series 3
or	(0,0)	={} for none.
ci3_u	(R,K*K)	upper bound for impulse response series 3
or	(0,0)	={} for none.
ci4_l	(R,K*K)	lower bound for impulse response series 4
or	(0,0)	={} for none.
ci4_u	(R,K*K)	upper bound for impulse response series 4
or	(0,0)	={} for none.
mainTitle	(1,1)	main title (""for none)
impulseNames	(K,1)	names of impulse variables

responseNames	(K,1)	names of response variables
row_sel	(N,1)	indices for rows to plot (N<=K)
or	(1,1)	0 for plotting all rows
col_sel	(M,1)	indices for columns to plot (M<=K)
or	(1,1)	0 for plotting all columns

Output

Graphics

Remarks

The impulse response functions (and the confidence bounds matrices) must be ordered row-wise, i.e. the first K columns in row $i+1$ in the input argument irf1 take the quantities of the first row of Φ_i and so forth.

Some ideas coming from graphic procedures of Sune Karlsson are used to display the main title.

Globals

__var_Note_StartTime
__var_div

Source

var_irap.src

var_plotIR4CI4NoSelection

Purpose

Generates plot of up to four impulse response functions and their confidence intervals.

Format

```
call var_plotIR4CI4NoSelection(irf1, irf2, irf3, irf4,
                               ci1_l, ci1_u, ci2_l, ci2_u,
                               ci3_l, ci3_u, ci4_l,
                               ci4_u, impulseNames,
                               responseNames, mainTitle);
```

Input

irf1	(R,K*K)	impulse response series 1
irf2	(R,K*K)	impulse response series 2
or	(0,0)	={} for none.
irf3	(R,K*K)	impulse response series 3
or	(0,0)	={} for none.
irf4	(R,K*K)	impulse response series 4
or	(0,0)	={} for none.
ci1_l	(R,K*K)	lower bound for impulse response series 1
or	(0,0)	={} for none.
ci1_u	(R,K*K)	upper bound for impulse response series 1
or	(0,0)	={} for none.
ci2_l	(R,K*K)	lower bound for impulse response series 2
or	(0,0)	={} for none.
ci2_u	(R,K*K)	upper bound for impulse response series 2
or	(0,0)	={} for none.
ci3_l	(R,K*K)	lower bound for impulse response series 3
or	(0,0)	={} for none.
ci3_u	(R,K*K)	upper bound for impulse response series 3
or	(0,0)	={} for none.
ci4_l	(R,K*K)	lower bound for impulse response series 4
or	(0,0)	={} for none.
ci4_u	(R,K*K)	upper bound for impulse response series 4
or	(0,0)	={} for none.
mainTitle	(1,1)	main title (""for none)

impulseNames (K,1)	names of impulse variables
responseNames(K,1)	names of response variables

Output

Graphics

Remarks

Sune Karlsson procedures are used to display the main title.

Globals

__var_Note_StartTime
__var_div

See

var_plotIR4CI4

Source

var_irap.src

var_plotIRCI

Purpose

Plots impulse response functions with confidence intervals.

Format

```
var_plotIRCI(irf, ci_l, ci_u);
```

Input

irf	(R,K*K)	Impulse response function.
ci_l	(R,K*K)	Lower bound for confidence interval.
	(1,1)	0 for none.
ci_u	(R,K*K)	Upper bound for confidence interval.
	(1,1)	0 for none.

Output

Graphics

Globals

__var_Note_StartTime
__var_div

See

var_plotIR4CI4

Source

var_irap.src

var_plotIRCI_Title

Purpose

Plots impulse response functions with confidence intervals and a main title.

Format

```
var_plotIRCI_Title(irf, ci_l, ci_u, titleString);
```

Input

irf	(R,K*K)	Impulse response function.
ci_l	(R,K*K)	Lower bound for confidence interval.
	(1,1)	0 for none.
ci_u	(R,K*K)	Upper bound for confidence interval.
	(1,1)	0 for none.
titleString	(1,1)	Title string.

Output

Graphics

Globals

__var_Note_StartTime
__var_div

See

var_plotIR4CI4

Source

var_irap.src

var_SetEstimationMethod

Purpose

Set estimation method for system estimation.

Format

```
y = var_SetEstimationMethod(model, method);
```

Input

`model`	`(Y,1) data buffer`	VAR or VEC
`method`	`(1,1) string`	Key for estimation method. Valid keys are `none`, `OLS`, `GLS`, `3SLS`, `SEQ`, `JOH`.

Output

`y (Z,1) data buffer`	VAR or VEC model with information about system estimation.

Remarks

Change in data buffer: `em_sys`

Globals

__var_Note_StartTime
__var_div

See

vml_InitVAR
var_EstimateModel

Source

`var_est.src`

vml__bootstrapIR

Purpose

Computes bootstrap impulse responses a given VAR-model.

Format

```
{irfe_hat, irfe_star, irfe_star_sd, iror_hat, iror_star,
iror_star_sd, new_seed_star, new_seed_star_sd} =
   vml__bootstrapIR(var, nob, seed, nob_sd, seed_sd,
                    compute_sd, ir_max);
```

Input

var	(N,1)	matrix	data buffer with model information
nob	(1,1)	matrix	number of bootstrap replications
seed	(1,1)	matrix	seed for drawing residuals (seed=0 for using no seed)
nob_sd	(1,1)	matrix	number of bootstrap replications for estimating the standard deviation
seed_sd	(1,1)	matrix	seed for drawing residuals when computing standard deviation (seed=0 for using no seed)
compute_sd	(1,1)	matrix	0:= do not compute bootstrap standard deviation. otherwise compute bootstrap standard deviation.
ir_max	(1,1)	matrix	maximum number impulse responses to compute

Output

irfe_hat	(1,(1+ir_max)*K*K)	matrix	estimated forecast error impulse responses
irfe_star	(nob,(1+ir_max)*K*K)	matrix	bootstrapped forecast error impulse responses
irfe_star_sd	(nob,(1+ir_max)*K*K)	matrix	bootstrapp standard deviation for irfe_star
iror_hat	(1,(1+ir_max)*K*K)	matrix	estimated orthogonal impulse responses

iror_star	(nob,(1+ir_max)*K*K)	matrix	bootstrapped orthogonal impulse responses
iror_star_sd	(nob,(1+ir_max)*K*K)	matrix	bootstrapp standard deviation for iror_star
new_seed_star	(1,1)	matrix	updated seed
new_seed_star_sd	(1,1)	matrix	updated seed

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_irbo.src

vml__bootstrapIRCheckInputs

Purpose

Not specified yet.

Format

```
Not specified yet.
```

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_irbo.src

vml__createDeterministicNames

Purpose

Creates default names for deterministic variables.

Format

```
names = vml__createDeterministicNames(n);
```

Input

`n (1,1)` Number of variables

Output

`names (n,1) char array` Default names.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_new.src

vml__createEndogenousNames

Purpose

Creates default names for endogenous variables.

Format

```
names = vml__createEndogenousNames(n);
```

Input

`n (1,1)` Number of variables

Output

`names (n,1) char array` Default names.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_new.src

vml__createErrorCorrectionNames

Purpose

Creates default names for variables in the cointegration relation.

Format

```
names = vml__createErrorCorrectionNames(n);
```

Input

`n (1,1)` Number of variables

Output

`names (n,1) char array` Default names.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_new.src

vml__createExogenousNames

Purpose

Creates default names for exogenous variables.

Format

```
names = vml__createExogenousNames(n);
```

Input

`n (1,1)` Number of variables

Output

`names (n,1) char array` Default names.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_new.src

vml__createMixedNames

Purpose

Creates default names for variables of time varying coefficients.

Format

```
names = vml__createMixedNames(n);
```

Input

`n (1,1)` Number of variables

Output

`names (n,1) char array` Default names.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_new.src

vml__createNamesVector

Purpose

Generates names vectors with lag structure on the fly.

Format

```
laggedNames = vml__createNamesVector(eqNames, pmin, pmax);
```

Input

eqNames	(M,1)	string array	Names of equations or variables
pmin	(1,1)	matrix	Minimum lag
pmax	(1,1)	matrix	Maximum lag

Output

laggedNames	(M*((pmax-pmin)+1),1)	string array	Variable names
	(0,0)	matrix	(empty)

Remarks

Mainly by other procedures used.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

Source

var_show.src

vml__divCVU

Purpose

Computes divisor for residual covariance matrix estimation.

Format

```
n = vml__divCVU(T, K, nu, nr);
```

Input

`T`	`(1,1) matrix`	Number of observations
`K`	`(1,1) matrix`	Number of equations
`nu`	`(1,1) matrix`	Number of parameters in unrestricted model
`nr`	`(1,1) matrix`	Number of parameters in restricted model

Output

`n`	`(1,1) matrix`	Devisor, depending on the global setting.

Remarks

If `__var_div`= 0 then $n = T$; if `__var_div`= 1 then $n = T - nr/K$.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

var_ls.src

vml__EstimateVARModel

Purpose
Estimates VAR model.

Format
```
y = vml__EstimateVARModel(model);
```

Input

`model (Y,1) data buffer` Defined VAR.

Output

`y (Z,1) data buffer` Estimated VAR.

Remarks
This procedure interprets the input argument as a VAR(p) model.
Changes in data buffer: `+ A0, A, B, C, F`

Globals
__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See
var_EstimateModel

Source
`var_est1.src`

vml__extractRowOfGLR

Purpose

Extracts from a linear restriction matrix the linear restrictions for a specified equation.

Format

```
Rnew = vml__extractRowOfGLR(Rold, idx, rType);
```

Input

`Rold`	`(M,N) matrix`	Restriction matrix
`idx`	`(1,1) matrix`	equation index
`rType`	`(1,1) matrix`	Determines type of restriction: 0: `Rold=r`, otherwise `Rold=R`

Output

`Rnew`	`(Q,R) matrix`	Restriction matrix for the specified equation.

Remarks

Helper method for `var__EstimateVECModel`.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

var__EstimateVECModel

Source

var_est2.src

vml__getRegMatrix

Purpose

Generates regression matrix from endogenous, exogenous and deterministic variables.

Format

```
Z = vml__getRegMatrix(y, x, d, py, px);
```

Input

y	(py+T,K) matrix	Observations for endogenous variables
x	(px+T,M) matrix	Observations for predetermined variables that enter the model with lags (see px)
d	(T,N) matrix	Observations for predetermined variables that do not enter the model with lags
py	(1,1) matrix	Number of presample values in y and lag number for endogenous variables
px	(1,1) matrix	Number of presample values in x and lag number for this variable

Output

Z	(T,py*K+(px+1)*M+N) matrix	Regression matrix with specified lag structure

Remarks

It can be the case that Z = {}.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

var_ls.src

vml__gls_lr

Purpose

GLS estimation with linear constraints.

Format

```
{parY, parX, parD, u, cv_par, cvu, msg} =
   vml__gls_lr(y, x, d, py, px, Ry, Ry_c, Rx, Rx_c, Rd, Rd_c);
```

Input

y	(py+T,K)	Observations for endogenous variables
x	(px+T,L)	Observations for predetermined variables that enter the model with lags (see px)
	(0,0)	if model does not include such variables x={}
d	(T,M)	Observations for predetermined variables that do not enter the model with lags
	(0,0)	if model does not include such variables d={}
py	(1,1)	Number of presample values in y and lag number for endogenous variables
px	(1,1)	Number of presample values in x and lag number for this variable
Ry	(K*K*py,M)	R-matrix for endogenous variables
	(0,0)	if none
Ry_c	(K*K*py,1)	r-matrix for endogenous variables
	(0,0)	if none
Rx	(L*K*(px+1),N)	R-matrix for exogenous variables
	(0,0)	if none
Rx_c	(L*K*(px+1),1)	r-matrix for exogenous variables
	(0,0)	if none
Rd	(M*K,0)	R-matrix for deterministic variables
	(0,0)	if none
Rd_c	(M*K,1)	r-matrix for deterministic variables
	(0,0)	if none

Output

parY	(K,K*py)	Coefficient matrices for endogenous variables
parX	(K,L*(px+1))	Coefficient matrices for exogenous variables
parD	(K,M)	Coefficient matrices for deterministic variables
u	(T,K)	Estimated residuals

cv_par	(W,W)	Covariance matrix of all coefficeints
cv_u	(K,K)	Covariance matrix of residuals
msg	(1,1)	Message string

Remarks

General linear restrictions are specified as

$$g = \text{vec}(G) = R\gamma + r,$$

where G is the parameter matrix of the endogenous, exogenous, or deterministic variable. It is not possible to set restrictions across different variable types.

cv_par is the estimated asymptotic covariance matrix of vec(parY|parX|parD) and has dimension (K*(K*py+L*(px+1)+M),K*(K*py+L*(px+1)+M)) For better performance use vml__gls_sr() in case of simple subset restrictions.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__gls_sr

Source

var_gls.src

vml__gls_sr

Purpose

GLS estimation with subset restrictions.

Format

```
{parY, parX, parD, u, cv_par, cvu, msg} =
   vml__gls_sr(y, x, d, py, px, sr_y, sr_x, sr_d);
```

Input

y	(py+T,K)	Observations for endogenous variables
x	(px+T,L)	Observations for predetermined variables that enter the model with lags (see px)
	(0,0)	If model does not include such variables x={}
d	(T,M)	Observations for predetermined variables that do not enter the model with lags
	(0,0)	If model does not include such variables d={}
py	(1,1)	Number of presample values in y and lag number for endogenous variables
px	(1,1)	Number of presample values in x and lag number for this variable
sr_y	(K,py*K)	Subset restriction matrix
	(1,1)	= 1 or = 0 as shortcut for matrix of ones and zeros.
	(0,0)	Empty matrix as shortcut for = 1
sr_x	(K,(px+1)*L)	Subset restriction matrix
	(1,1)	= 1 or = 0 as shortcut for matrix of ones and zeros.
	(0,0)	Empty matrix as shortcut for = 1
sr_d	(K,M)	Subset restriction matrix
	(1,1)	= 1 or = 0 as shortcut for matrix of ones and zeros.
	(0,0)	Empty matrix as shortcut for = 1

Output

parY	(K,K*py)	Coefficient matrices for endogenous variables
parX	(K,L*(px+1))	Coefficient matrices for exogenous variables
parD	(K,M)	Coefficient matrices for deterministic variables
u	(T,K)	Estimated residuals

cv_par	(W,W)	Covariance matrix of all coefficeints
cv_u	(K,K)	Covariance matrix of residuals
msg	(1,1)	Message string

Remarks

The subset restrictions are specified such that a 1 indicates the inclusion of the respective variable (and estimation of the coefficient), and a 0 indicates to exclude the respective variable from the estimation algorithm. For more information on output parameters see also remarks in vml__gls_lr.

This procedure traps problems while matrix inversion. The first returned argument should be checked against scalerr to see whether problems occurred during calling this procedure.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__gls_lr

Source

var_gls.src

vml__johansenApproach

Purpose

Estimates VEC-model with the Johansen approach.

Format

```
{alpha, beta, beta_d, G, C, u, cvu, cv_P, cv_alpha, cv_G} =
   vml__johansenApproach(y, pdy, d_ec, d_vec, r);
```

Input

y	(pdy+T,K)	Observations for endogenous variables
pdy	(1,1)	Number of presample values in y and
d_ec	(pdy+T,N)	Observations for deterministic variables that enter the cointegration relation
d_vec	(T,M)	Observations for exogenous variables
r	(1,1)	Cointegration rank

Output

alpha	(K,r)	Loading matrix
beta	(r,K)	Normalized cointegration vectors cointegration relation.
beta_d	(r,N)	Coefficient matrix for deterministic variables in the cointegration relation.
G	(K,(pdy-1)*K)	Coefficient matrix for endogenous variables (differences)
C	(K,M)	Coefficient matrix for exogenous variables
u	(T,K)	Residuals
cvu	(K,K)	Residual covariance matrix. See Equation (6.12) of Johansen (1995)
cv_P	(K,K)	Covariance matrix of alpha*beta-matrix. See Theorem 13.7 of Johansen (1995)
cv_alpha	(K,r)	Covariance matrix of alpha-matrix. See Theorem 13.3 of Johansen (1995)
cv_G	(K,(pdy-1)*K)	Covariance matrix of G matrix. See Theorem 13.5 of Johansen (1995)

Remarks

Procedure implements is oriented on the theory laid out in Johansen (1995).

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

var_joh.src

vml__lagNames

Purpose

Adds literal that indicates a lag to the name array.

Format

```
Not specified yet.
```

Input

`n`	`(M,N) string array`	Name array
`lag`	`(1,1) matrix`	The lag to be added

Output

`r`	`(M,N) string array`	Name array with lag indication. No lag literal is added, if `lag=0`.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_new.src`

vml__lagObservations

Purpose

Helper method for estimation procedures.

Format

```
y = vml__lagObservations(x, px, startLag, endLag);
```

Input

x	(px+T,K)	matrix	Observation matrix
px	(1,1)	matrix	Number of presample values
startLag	(1,1)	matrix	First lag for lag structure
endLag	(1,1)	matrix	Last lag for lag structure

Output

y	(T,(endLag-startLag+1)*K)	matrix	Observation matrix with lag-structure

Remarks

If

$$Y = \begin{bmatrix} y_{1-p} \\ y_{2-p} \\ \vdots \\ y_T \end{bmatrix}$$

is the matrix of observations, the i-th row of the output matrix has the structure

$$\left[y_i, y_{i-1}, \ldots, y_{i-(endLag-startLag+1)}\right]$$

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

var_ls.src

vml__mergeRConstraints

Purpose

Merges general linear constraints of two variable groups.

Format

```
{R, R_c} =
   vml__mergeRConstraints(ny, nx, Ry, Ry_c, Rx, Rx_c);
```

Input

ny	(1,1)	number of parameters in unrestricted model for that variable
nx	(1,1)	number of parameters in unrestricted model for that variable
Ry	(ny,M)	R-matrix for variable group y
	(0,0)	if none
Ry_c	(ny,1)	r-matrix for variable group y
	(0,0)	if none
Rx	(nx,N)	R-matrix for variable group x
	(0,0)	if none
Rx_c	(nx,1)	r-matrix for variable group x
	(0,0)	if none

Output

R	(ny+nx,M+N)	R-matrix for variable group [y—x]
R_c	(ny+nx,1)	r-matrix for variable group [y—x]

Remarks

General linear restrictions are specified as

$$g = \text{vec}(G) = R\gamma + r,$$

where G is the parameter matrix of the endogenous, exogenous, or deterministic variable. It is not possible to set restrictions across different variable types.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__ols
vml__gls

Source

var_ls.src

vml__ols_lr

Purpose

OLS estimation with general linear constraints.

Format

```
{parY, parX, parD, u, cv_par, cvu} =
   vml__ols_lr(y, x, d, py, px, Ry, Ry_c, Rx, Rx_c, Rd, Rd_c);
```

Input

y	(py+T,K)	Observations for endogenous variables
x	(px+T,L)	Observations for predetermined variables that enter the model with lags (see px)
d	(T,M)	Observations for predetermined variables that do not enter the model with lags
py	(1,1)	Number of presample values in y and lag number for endogenous variables
px	(1,1)	Number of presample values in x and lag number for this variable
Ry	(K*K*py,M)	R-matrix for endogenous variables
	(0,0)	if none
Ry_c	(K*K*py,1)	r-matrix for endogenous variables
	(0,0)	if none
Rx	(L*K*(px+1),N)	R-matrix for exogenous variables
	(0,0)	if none
Rx_c	(L*K*(px+1),1)	r-matrix for exogenous variables
	(0,0)	if none
Rd	(M*K,0)	R-matrix for deterministic variables
	(0,0)	if none
Rd_c	(M*K,1)	r-matrix for deterministic variables
	(0,0)	if none

Output

b_hat	(K,N)	Estimated coefficient matrices
	or scalmiss	
cv_b_hat	(K*N,K*N)	Estimated asymptotic covariance matrix
	or scalmiss	
u_hat	(T,K)	Estimated residuals
cv_u_hat	(K,K)	Estimated covariance matrix of residuals

Remarks

This procedure takes three different variable types:

y Endogenous variables

x Exogenous variables (lagged)

d Exogenous variables (unlagged).

The regression model is

$$\begin{aligned} y_t &= A_1 y_{t-1} + \ldots + A_p y_{t-p} \\ &\quad + B_0 x_t + B_1 x_{t-1} + \ldots + B_q x_{t-q} \\ &\quad + C d_t + u_t \end{aligned}$$

subject to general linear constraints in the form of

$$g = \text{vec}(G) = R\gamma + r,$$

where G is the parameter matrix of the endogenous, exogenous, or deterministic variable. It is not possible to set restrictions across different variable types.

The estimated coefficient matrix `b_hat` is organized such that `b_hat`= $[\widehat{A_1} : \ldots : \widehat{A_p} : \widehat{B_0} : \widehat{B_1} : \ldots : \widehat{B_q} : \widehat{C}]$, where $\widehat{\cdot}$ is the OLS estimate of the respective quantity.

`cv_b_hat` is the estimated asymptotic covariance matrix of `vec(b_hat)`.

For better performance use `vml__ols_sr()` in case of simple subset restrictions.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__ols_sr

Source

`var_ols.src`

vml__ols_sr

Purpose

OLS estimation with subset restrictions.

Format

```
{parY, parX, parD, u, cv_par, cvu} =
   vml__ols_sr(y, x, d, py, px, sr_y, sr_x, sr_d);
```

Input

y	(py+T,K)	observations for endogenous variables
x	(px+T,L)	observations for predetermined variables that enter the model with lags (see px)
	(0,0)	if model does not include such variables x={}
d	(T,M)	observations for predetermined variables that do not enter the model with lags
	(0,0)	if model does not include such variables d={}
py	(1,1)	number of presample values in y and lag number for endogenous variables
px	(1,1)	number of presample values in x and lag number for this variable
sr_y	(K,py*K)	Subset restriction matrix
	(1,1)	= 1 or = 0 as shortcut for matrix of ones and zeros.
	(0,0)	Empty matrix as shortcut for = 1
sr_x	(K,(px+1)*L)	Subset restriction matrix
	(1,1)	= 1 or = 0 as shortcut for matrix of ones and zeros.
	(0,0)	Empty matrix as shortcut for = 1
sr_d	(K,M)	Subset restriction matrix
	(1,1)	= 1 or = 0 as shortcut for matrix of ones and zeros.
	(0,0)	Empty matrix as shortcut for = 1

Output

parY	(K,py*K)	Estimated coefficient matrices of endogenous variables
parX	(K,(px+1)*L)	Estimated coefficient matrices of exogenous variables

parD	(K,M)	Estimated coefficient matrices of deterministic variables
u	(T,K)	Estimated residuals
cv_par	(K*N,K*N)	Estimated asymptotic covariance matrix of vec($parY : parX : parD$)
cvu	(K,K)	Estimated covariance matrix of residuals

Remarks

For general linear constraints use vml__ols_lr instead.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__ols_lr

Source

var_ols.src

vml__prepareLS

Purpose

Prepare OLS or GLS estimation.

Format

```
{y, x, R, R_c} =
   vml__prepareLS(y, x, d, py, px, Ry, Ry_c, Rx, Rx_c, Rd,
                  Rd_c);
```

Input

y	(py+T,K)	Observations for endogenous variables
x	(px+T,L)	Observations for predetermined variables that enter the model with lags (see px)
	(0,0)	if model does not include such variables x={}
d	(T,M)	Observations for predetermined variables that do not enter the model with lags
	(0,0)	if model does not include such variables d={}
py	(1,1)	Number of presample values in y and lag number for endogenous variables
px	(1,1)	Number of presample values in x and lag number for this variable
Ry	(K*K*py,M)	R-matrix for endogenous variables
	(0,0)	if none
Ry_c	(K*K*py,1)	r-matrix for endogenous variables
	(0,0)	if none
Rx	(L*K*(px+1),N)	R-matrix for exogenous variables
	(0,0)	if none
Rx_c	(L*K*(px+1),1)	r-matrix for exogenous variables
	(0,0)	if none
Rd	(M*K,0)	R-matrix for deterministic variables
	(0,0)	if none
Rd_c	(M*K,1)	r-matrix for deterministic variables
	(0,0)	if none

Output

y	(T,K)	The lhs observations (regressand)

x	(T,K*py+L*(px+1)+M)	The regressor matrix. This matrix might be empty (x={}) if the specification excludes predetermined variables.
R	(K*(K*py+L*(px+1)+M),N)	R-matrix for the rewritten estimation problem
R_c	(K*(K*py+L*(px+1)+M),K)	r-matrix for the rewritten estimation problem

Remarks

General linear restrictions are specified as

$$g = \text{vec}(G) = R\gamma + r,$$

where G is the parameter matrix of the endogenous, exogenous, or deterministic variable. It is not possible to set restrictions across different variable types.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__ols
vml__gls

Source

var_ls.src

vml__quantiles

Purpose

Computes symmetric quantiles from given discrete distribution function and coverage probability.

Format

```
{t1, t2} = vml__quantiles(L, c);
```

Input

`L B x M matrix`	Discrete distribution functions of `M` random variables
`c 1 x 1 matrix`	Coverage, `0<c<1`

Output

`t1 1 x M matrix`	lower confidence bound
`t2 1 x M matrix`	upper confidence bound

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_boot.src`

vml__recserVAR

Purpose

Computes a multivariate autoregressive recursive series.

Format

```
y =
   vml__recserVAR(A0, A, B, C, F, mx_c, mx_tf, y0, u, x, d,
                  z);
```

Input

A0	(K,K)	matrix	Left hand side structural coefficient matrix
A	(K,K*p)	matrix	Right hand side structural coefficient matrix
B	(K,L*(q+1)	matrix	Coefficient matrix of exogenous variables
C	(K,M)	matrix	Coefficient matrix of deterministic variables
F	(K,N)	matrix	Coefficient matrix of deterministic variables
mx_c	(N,4)	matrix	Rule for mixed variables
mx_tf	(T,0)	matrix	Time function for mixed variables
y0	(p,K)	matrix	Start values
u	(T,K)	matrix	Residuals
x	(q+T,L)	matrix	Exogenous variables
d	(T,M)	matrix	Deterministic variables
z	(T,N)	matrix	Mixed variables

Output

y	(py+T,K)	matrix	Multivariate autoregressive recursive series

Remarks

This procedure considers variables that were constructed from endogenous and deterministic variables (mixed or time varying parameters). The VAR model behind is defined in Equation (D.2):

$$\widetilde{A}_0 y_t = \widetilde{A} Y_{t-1} + \widetilde{B} X_t + \widetilde{C} d_t + \widetilde{F} Z_t + u_t$$

See Section D.2.2 for more details.

Globals

__var_Note_StartTime
__var_div

__var_Note_StartTime
__var_div

Source

var_boot.src

vml__subsetToGeneralRestriction

Purpose

Transforms subset restriction matrix to general linear restriction matrix.

Format

```
y = vml__subsetToGeneralRestriction(x);
```

Input

`x (K,L) matrix` subset restriction matrix

Output

`y (K*L,M) matrix` General linear restriction matrix.

Remarks

A subset restriction matrix is defined as a matrix of ones and zeros. It has the same dimension as the associated parameter matrix A. A zero element in the subset restriction matrix indicates that the respective parameter in the parameter matrix is set to zero in the estimation procedure (the respective variable drops out of the regression model).

A general linear restriction matrix R is defined in the equation

$$\text{vec}(A) = R\gamma + r$$

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

var_rest.src

vml__syncSRMatrix

Purpose

Synchronizes subset restriction matrix with respect to dimension.

Format

```
y = vml__syncSRMatrix(x, K, n);
```

Input

x	(K,n) matrix	Proposed subsetrestriction matrix
	(1,1) matrix	x=0 or x=1
	(0,0) matrix	Equal to x=1
K	(1,1) matrix	Number of equations
n	(1,1) matrix	Number of variables in each equation

Output

x	(K,n) matrix	subset restriction matrix

Remarks

It is guaranteed that the returned matrix has the correct dimension.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

var_ls.src

vml__timeVaryingParameter

Purpose

Computes time variying coefficient at a specified date.

Format

```
AA = vml__timeVaryingParameter(A, Atf, tf, c, idx);
```

Input

`A`	`(M,N) matrix`	Basis coefficient matrix
`Atf`	`(M,L) matrix`	Time variying coefficient matrix
`tf`	`(T,L) matrix`	Time function
`c`	`(1,L) matrix`	Indices between 1 and N, specifies the variable
`idx`	`(1,1) matrix`	Time index (runs from 1 to T)

Output

`AA`	`(M,N) matrix`	Combination of `A`, `Atf` and `tf`, the time varying coefficient matrix at the specified time.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__recserVAR

Source

`var_boot.src`

vml_ComputeBootstrapDraw

Purpose

Generates a bootstrap time series for a VAR model.

Format

{y, s1} = vml_ComputeBootstrapDraw(varHat, seed);

Input

varHat	(N,1) data buffer	Estimated VAR model
seed	(1,1) matrix	Seeding value, $0 < seed < (2^{31}) - 1$, otherwise it is ignored

Output

y	(py+T,K) matrix	Bootstrap time series
s1	(1,1) matrix	Updated seed

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_residualBootstrap
vml_InitVAR
var_EstimateModel

Source

var_boot.src

vml_GetCointRank

Purpose

Returns the cointegration rank of the model.

Format

```
r = vml_GetCointRank(model);
```

Input

`model (M,1) data buffer` VAR or VECM model

Output

`r (1,1) matrix` cointegration rank

Remarks

The input argument `model` defines a K-dimensional VAR(p) model. This procedure searches for some indication that a cointegration rank r has been set. This could have been done with `vml_SetCointRelation()`.

Note that if $r = 0$ Δy_t has a stable VAR($p-1$) representation and for $r = K$ y_t is a stable VAR(p). This is considered in the estimation procedure.

It is guaranteed that $0 \leq r \leq K$. If for some reason this condition is not satisfied the procedure terminates with an errorlog. This means that it is not necessary to check the returned value for plausibility.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_InitVAR
vml_SetCointRelation

Source

var_rest.src

vml_infrv

Purpose

Return a matrix with all infinities (-/+) set to a given value.

Format

```
y = vml_infrv(x, a);
```

Input

`x (M,N) matrix`	Argument matrix
`a (1,1) matrix`	Value to be set instead of all -infinite or +infinite found in `x`

Output

`y (M,N) matrix`	It is guaranteed that `y` does not contain any +/- infinite values

Remarks

Calls `vml_setINF`

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_setINF

Source

`var_tool.src`

vml_InitVAR

Purpose

Defines a VAR model.

Format

```
var = vml_InitVAR(y, py, Ny, x, px, Nx, d, Nd);
```

Input

`y`	`(py+T,K)`	`matrix`	Time series of endogenous variables
`py`	`(1,1)`	`matrix`	Number of lagged endogenous variables ($:= p$)
`Ny`	`(K,1)`	`string array`	Names of endogenous variables
	`(0,0)`		`Ny={}`
`x`	`(px+T,L)`	`matrix`	Time series of exogenous variables
	`(0,0)`		`x={}` if no exogenous variables are defined in the system
`px`	`(1,1)`	`matrix`	Number of lagged exogenous variables (ignored if `x={}`) ($:= q$)
`Nx`	`(L,1)`	`string array`	Names of exogenous variables
	`(0,0)`		`Nx={}`
`d`	`(py+T,M)`	`matrix`	Time series of deterministic variables
	`(0,0)`		`d={}` if no deterministic variables are defined in the system
`Nd`	`(M,1)`	`string array`	Names of deterministic variables
	`(0,0)`		`Nd={}`

Output

`var (N,1) data buffer` VAR model.

Remarks

This procedure collects all observations necessary to estimate the following VAR model

$$\begin{aligned} y_t &= A_1 y_{t-1} + \ldots + A_p y_{t-p} \\ &\quad + B_0 x_t + B_1 x_{t-1} + \ldots + B_q x_{t-q} \\ &\quad + C d_t + u_t \qquad\qquad t = 1, \ldots, T \end{aligned}$$

It is not necessary to provide the variable names. Default names are generated in this case.

This procedure generates the following fields [optional fields] in the returned data buffer: `y, py [,Ny, x, px, Nx, d, Nd]`

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_init.src`

vml_print_ndpchk

Purpose

Examines exceptions have been generated by the numeric data processor and prints some contextual message.

Format

```
call vml_print_ndpchk(msg);
```

Input

`msg (1,1) string` Contextual string message

Output

`Message on screen`

Remarks

See Gauss function `ndpchk` for more details.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_tool.src`

vml_regroup

Purpose

Row vector to matrix representation.

Format

```
y = vml_regroup(x, N);
```

Input

`x (1xm) matrix`	The row vector with `m=N*k`
`N (1x1) matrix`	Number of rows in the target matrix

Output

`y (N,k) matrix`	Transformed matrix.

Remarks

This procedure is the reverse to the `vec` operator given the information about the dimension of the original matrix. The elements of `x` are stored column wise in `y`.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_boot.src`

vml_residualBootstrap

Purpose

Generates a bootstrap time series for a VAR model.

Format

```
{y, s1} =
   vml_residualBootstrap(A0, A, B, C, F, mx_c, mx_tf, y0, u,
                         x, d, z, seed);
```

Input

A0	(K,K)	matrix	Left hand side structural coefficient matrix
A	(K,K*p)	matrix	Right hand side structural coefficient matrix
B	(K,L*(q+1)	matrix	Coefficient matrix of exogenous variables
C	(K,M)	matrix	Coefficient matrix of deterministic variables
F	(K,N)	matrix	Coefficient matrix of deterministic variables
mx_c	(N,4)	matrix	Rule for mixed variables
mx_tf	(T,0)	matrix	Time function for mixed variables
y0	(p,K)	matrix	Start values
u	(T,K)	matrix	Residuals
x	(q+T,L)	matrix	Exogenous variables
d	(T,M)	matrix	Deterministic variables
z	(T,N)	matrix	Mixed variables
seed	(1,1)	matrix	Seeding value, $0 < seed < (2^{31}) - 1$, otherwise it is ignored

Output

y	(p+T,K)	matrix	Bootstrap time series
s1	(1,1)	matrix	Updated seed

Remarks

See Section D.2.3 for defining mixed variables.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime

__var_div

See

vml_ComputeBootstrapDraw

Source

var_boot.src

vml_residualBootstrap_prepare

Purpose

Extracts data for other procedures.

Format

```
{A0, A, B, C, F, mx_cr, mx_tf, y0, u, x, d, z} =
   vml_residualBootstrap_prepare(varHat);
```

Input

`varHat (N,1) data buffer` Estimated VAR model

Output

`A0`	`(K,K)`	`matrix`	Left hand side structural coefficient matrix
`A`	`(K,K*p)`	`matrix`	Right hand side structural coefficient matrix
`B`	`(K,L*(q+1)`	`matrix`	Coefficient matrix of exogenous variables
`C`	`(K,M)`	`matrix`	Coefficient matrix of deterministic variables
`F`	`(K,N)`	`matrix`	Coefficient matrix of deterministic variables
`mx_c`	`(N,4)`	`matrix`	Rule for mixed variables
`mx_tf`	`(T,0)`	`matrix`	Time function for mixed variables
`y0`	`(p,K)`	`matrix`	Start values
`u`	`(T,K)`	`matrix`	Residuals
`x`	`(q+T,L)`	`matrix`	Exogenous variables
`d`	`(T,M)`	`matrix`	Deterministic variables
`z`	`(T,N)`	`matrix`	Mixed variables

Remarks

Helper method to prepare the input arguments for `vml_residualBootstrap()`.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_residualBootstrap

Source

`var_boot.src`

vml_SetCointRelation

Purpose

Sets r cointegration relation(s) and implicitly the cointegration rank.

Format

```
var = vml_SetCointRelation(var, beta, beta_d);
```

Input

`model`	`(N,1) data buffer`	Defined VEC or VAR model
`beta`	`(r,K) matrix`	Cointegration vector(s)
`beta_d`	`(r,M) matrix`	Parameters for deterministic variables entering β

Remarks

Changes in data buffer:
+beta_x[, beta_d_x]

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

var_rest.src

vml_SetEndogenousVariables

Purpose

Updates observations for endogenous variables.

Format

```
var1 = vml_SetEndogenousVariables(var0, y1);
```

Input

var0 (1xm)	data buffer	The defined VAR or VECM.
y1	(p+TxK) matrix	The new observations.

Output

var1 (Z,1)	data buffer	The updated VAR.

Remarks

All data buffer elements depending on the endogenous variables are recomputed, estimation results deleted.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml__bootstrapIR

Source

var_boot.src

vml_setINF

Purpose

Return a matrix with all infinities (-/+) set to a given value.

Format

```
y = vml_setINF(x, a);
```

Input

`x (M,N) matrix`	Argument matrix
`a (1,1) matrix`	Value to be set instead of all -infinite or +infinite found in `x`

Output

`y (M,N) matrix`	It is guaranteed that `y` does not contain any +/- infinite values

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_tool.src`

vml_SetModelExtensions

Purpose

Adds time dependent variables to the model.

Format

```
y = vml_SetModelExtensions(model, mx_c, mx_tf);
```

Input

`model`	`(M,1) data buffer`	Defined VAR or VEC model
`mx_c`	`(0,4) matrix`	where `mx_c[.,1]` specifies the variable type (see Remark) `mx_c[.,2]` specifies the variable index `mx_c[.,3]` specifies the lag `mx_c[.,4]` specifies the column index of the time function in `mx_tf` (`mx_c[.,4]<1` if no time function is needed)
`mx_tf`	`(T,P) matrix`	`P<=T`. If `mx_c[i,3]<1` for all `i`, `mx_tf := {}`

Output

`model`	`(N,1) data buffer`	Extended model.

Remarks

This procedure allows to combine deterministic variables (v_t) and stochastic variables (w_t) multiplicatively:

$$z_t = v_t w_t.$$

It is required that w_t is contained in the endogenous or exogenous variables. The advantage over adding the variable z_t directly to the model is that the impulse response analysis and the bootstrap can process that information.

The following variable types are valid:

	coding
endogenous	enl (level) end (differenced)
exogenous	ex
EC-term	ec
determin	de

Changes in data buffer:

`+ mx_c[, mx_tf], Z`

Globals

__var_Note_StartTime

__var_div
__var_Note_StartTime
__var_div

See
vml_InitVAR

Source
`var_mx.src`

vml_SetRestrictions

Purpose

Imposes linear restrictions on model parameters.

Format

```
modelR = vml_SetRestrictions(model, rx, name);
```

Input

`model`	`(Z,1) data buffer`	A VAR or VEC model
`rx`	`(M,N) matrix`	Restriction matrix
`name`	`(1,1) matrix`	name for restriction matrix
	`(1,1) string`	name for restriction matrix

Output

`modelR`	`(U,1) data buffer`	A VAR or VEC model with the given linear constraint.

Remarks

Call this procedure after `vml_InitVAR()`.
It is possible to impose general linear restrictions of the form $\beta = R\gamma + r$. Additionally, one can impose subset restrictions by specifying a matrix of ones and zeros for the respective parameter matrix (1=estimate the respective coefficient, 0=restrict this coefficient to zero). See Section D.2.4 for more details. Change in data buffer:
For linear restrictions `+R_XX[, R_XX_C]`,
and for subset restrictions `+S_XX`

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_InitVAR

Source

`var_rest.src`

vml_ShowData

Purpose

Prints information about model to the screen.

Format

```
call vml_ShowData(model, showGraphs);
```

Input

`model (M,1) data buffer`	VEC or VAR model specified for estimation or already estimated

Output

`Screen output`

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_show.src`

vml_showDim

Purpose

Prints dimension of argument on screen.

Format

```
call vml_showDim(x);
```

Input

`x (M,N) matrix, string [array]` Argument

Output

`Screen output`

Example

The code

```
y=ones(13,2);
call vml_showDim(y);
```

returns

```
Dimension:  [13,2]
```

to the screen.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_tool.src`

vml_ShowEstimationResults

Purpose

Prints estimation results to the screen.

Format

```
call vml_ShowEstimationResults(model);
```

Input

`model (M,1) data buffer` Estimated VEC or VAR model

Output

`Screen output`

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

Source

`var_show.src`

vml_VAR_For_IRA

Purpose

Transforms a VAR model for the impulse response analysis.

Format

```
newVAR = vml_VAR_For_IRA(model);
```

Input

`model (N,1) data buffer` Estimated VAR model.

Output

`newVAR (M,1) data buffer` Transformed VAR model.

Remarks

The current implementation does nothing with the argument VAR model.

Redefine this procedure if you want to compute impulse responses from a VAR model which is based on a transformed version, for example you might want to include identities before conducting the structural analysis. This procedure is called in all bootstrap confidence interval routines.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_InitVAR

Source

var_V4I.src

vml_vdel

Purpose

Modified `vdel(dbuf, xname)`. Also deletes string array elements (if added with vml_vput()).

Format

```
x = vml_vdel(model, keyList);
```

Input

`dbuf` `(Kx1) matrix` databuffer constructed with vput

`xname (1x1) string` name of variable

Output

`x (M,N) matrix or string array` data

Remarks

In order to delete an element of type string array it must have been putted using `vml_vput()`.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_vput

Source

`var_tool.src`

vml_VeRead

Purpose

Modified `vread(dbuf, name)`. Checks existence of `name` before returning.

Format

```
x = vml_VeRead(dbuf, name);
```

Input

`dbuf (Kx1) matrix` databuffer constructed with vput

`name (1x1) string` name of variable

Output

`x (M,N) string or matrix` if `name` is in `dbuf`

`(0,0)` if `name` is not in `dbuf`

Remarks

If the element `name` is not found in `dbuf` an empty symbol is returned. In such a case `vread()` simply brings out an error message.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_vread
vml_vput

Source

`var_tool.src`

vml_vput

Purpose

Modified `vput(dbuf, x, xname)`. Also works for string array arguments. Empty symbols (rows==0) are not written to the data buffer.

Format

```
dbufNew = vml_vput(dbuf, x, xname);
```

Input

`dbuf`	`(Kx1) matrix`	databuffer constructed with vput
`x`	`(M,N) matrix or string array`	data
`xname`	`(1x1) string`	name of variable

Output

`dbufNew (L,1) matrix` new data buffer

Remarks

In order to read the string array from the data buffer one must use `vml_vread()`.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_vread
vml_VeRead

Source

`var_tool.src`

vml_vread

Purpose

Modified `vread(dbuf, xname)`. Also reads string array elements (if added with vml_vput()).

Format

```
x = vml_vread(dbuf, xname);
```

Input

`dbuf` `(Kx1) matrix` databuffer constructed with vput

`xname (1x1) string` name of variable

Output

`x (M,N) matrix or string array` data

Remarks

In order to return an element of type string array it must have been putted using `vml_vput()`.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div

See

vml_vput
vml_VeRead

Source

var_tool.src

xxx_3sls

Purpose

Computes three-stage-least-squares estimates.

Format

{coeff, cv_coeff, u, cv_u, msg} = xxx_3sls(yx, sr);

Input

yx (T,K+M) matrix observations

sr (K,K+M) matrix subset restrictions

Output

coeff	(K,K+M)	matrix	Model coefficients.
cv_coeff	(K*(K+M),K*(K+M))	matrix	Covariance matrix of model coefficients.
u	(T,K)	matrix	Model residuals.
cv_u	(K,K)	matrix	Covariance matrix of model residuals.
msg	(1,1)	string	Message about estimation details.

Remarks

The model contains K equations and M predetermined variables. The sr matrix contains 1s and 0s only, a 1 indicates to include the respective variable. The K diagonal elements of sr are ignored.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div
__var_iter

See

xxx__i3sls

Source

var_3sls.src

xxx__i3sls

Purpose

Computes iterated three-stage-least-squares estimates.

Format

{coeff, cv_coeff, u, cv_u, msg} = xxx__i3sls(obs, mod, noi);

Input

obs	(T,K+M)	matrix	Endogenous and predetermined variables
mod	(K+M,K)	matrix	endogenous and predetermined variables
noi	(1,1)	scalar	number of iterations 0,1,2,...

Output

coeff	(K,K+M)	matrix	Model coefficients.
cv_coeff	(K*(K+M),K*(K+M))	matrix	Covariance matrix of model coefficients.
u	(T,K)	matrix	Model residuals.
cv_u	(K,K)	matrix	Covariance matrix of model residuals.
msg	(1,1)	string	Message about estimation details.

Remarks

cols(mod)=number of equations rows(obs)=number of observations
The $(K+i)$-th row of mod may be filled with zeros. This means that the i-th predetermined variable is not included in ANY of the K equations. Therefore the i-th variable is removed before doing the estimation (removing $K+i$-th column of obs and $K+i$-th row of mod).
The j-th column of mod may be filled with zeros. This means that the j-th equation contains no estimateable parameters. Therefore the j-th equation is removed before doing the estimation (removing the j-th column of obs and the j-th row and column of mod).

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div
__var_iter

Source

`var_3sls.src`

xxx__i3sls_cv

Purpose

Computes model residuals and covariance matrix of model residuals.

Format

{u, cv_u} = xxx__i3sls_cv(obs, mod_h, mod);

Input

obs	(T,K+M) matrix	Endogenous and predetermined variables.
mod_h	(K+M,K) matrix	Model coefficients.
mod	(K+M,K) matrix	Matrix of subset restrictions.

Output

u	(T,K) matrix	Model residuals.
cv_u	(K,K) matrix	Covariance matrix of model residuals.

Globals

__var_Note_StartTime
__var_div
__var_Note_StartTime
__var_div
__var_iter

See

vml__divCVU

Source

var_3sls.src

Lebenslauf

Name:	Alexander Benkwitz
09/1990 - 08/1991	Studium an der Hochschule für Ökonomie, Berlin-Karlshorst
09/1991 - 03/1996	Studium an der Humboldt-Universität zu Berlin in der Fachrichtung Betriebswirtschaftslehre
09/1993 - 03/1995	Studium an der Strathclyde University, Glasgow in der Fachrichtung Ökonometrie
06/1996 - 12/1996	Graduiertenkolleg "Angewandte Mikroökonomie", Humboldt-Universität zu Berlin,
04/1997 - 03/2002	Wissenschaftlicher Mitarbeiter an der Humboldt-Universität zu Berlin, Lehrstuhl Prof. Lütkepohl, Sonderforschungsbereich 373

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit ohne fremde Hilfe verfaßt und nur die angegebene Literatur und Hilfsmittel verwendet zu haben. Die vorliegende Dissertation war weder vollständig noch in Teilen Gegenstand einer früheren Begutachtung.
Ich bezeuge durch meine Unterschrift, dass meine Angaben über die bei der Abfassung meiner Dissertation benutzten Hilfsmittel, über die mir zuteil gewordene Hilfe sowie über frühere Begutachtungen meiner Dissertation in jeder Hinsicht der Wahrheit entsprechen.

Alexander Benkwitz
30. April 2002