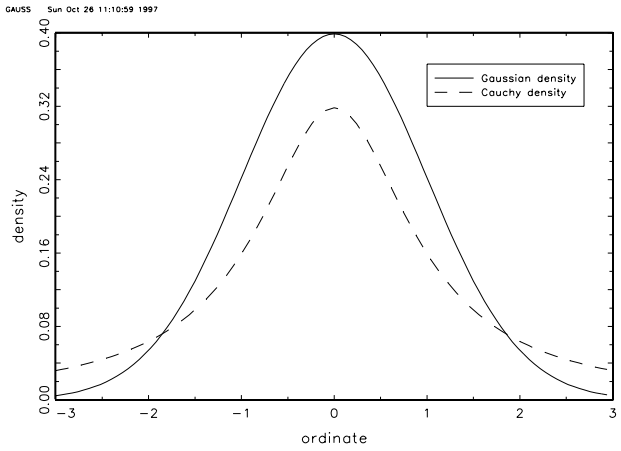
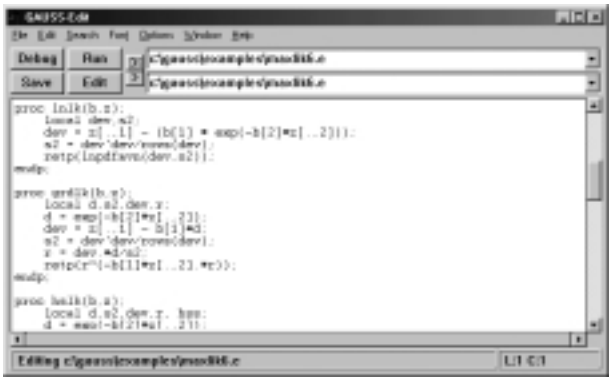
A short introduction to GAUSS for Windows

Koning, Ruud H.

A Short Introduction to GAUSS for Windows

Ruud H. Koning





A Short Introduction to GAUSS for Windows

Ruud H. Koning

Second, corrected printing November 1999

ISBN 90-804955-1-4

CONTENTS

CHAPTER 1

INTRODUCTION

During the last decade, powerful desktop computers have become available to most researchers. Mainframe computers have been replaced by personal computers and user friendly numerical programs have replaced old FORTRAN compilers and punch cards. As far as applications are concerned, there is a trend from writing a program for each problem to the use of professional software like statistical programs (SPSS, SAS), spreadsheet programs (Excel, Quattro), and matrix programming languages (MatLab, GAUSS). Using a matrix programming language is attractive compared to programming in a language like C++ or Pascal because the user does not have to implement the datatype 'matrix of numbers'. Moreover, many convenient functions are available and this makes writing programs easier. In this booklet we focus on one particular matrix programming language: GAUSS. The GAUSS-programming environment does not create stand-alone executables. Instead, a GAUSS program is compiled to pseudo-code and this pseudo-code is interpreted by the GAUSS interpreter.

GAUSS is available for different platforms. It was developed originally for MSDOS-based computers, and it has been ported to UNIX during the last few years. In this book we discuss the Windows-version of GAUSS[1]. This version is a 32-bits program that runs under Windows95, Windows98, and Windows NT. The development of this program is still ongoing, but the Windows version (3.35) that is available at the time of writing of this book is sufficiently developed. Most commands and examples will also run on the other platforms.

GAUSS is a convenient matrix programming language for doing econometric research. It is somewhat less suitable as a data management program, statistical programs such as SPSS or SAS are more suited to that task. A major advantage of GAUSS to programming languages is the availability

[1] All examples and graphs in this book were made using version 3.2.35.

of libraries for specific tasks and the availability of many built-in functions useful to econometricians. One library especially useful to econometricians is the maximum likelihood library that can be used for optimization of log-likelihood functions. A complete list of all libraries is given in Appendix B. Apart from the commercial libraries, one can also find program code and complete libraries on the internet. Moreover, most authors of scientific papers are willing to share their code.

There is some GAUSS support on the internet. First of all, there is the GAUSS mailing list (subscribe by sending an email[2] with contents `subscribe gaussians` to `majordomo@eco.utexas.edu`) which is archived in searchable html-format at `www.rhkoning.com/gauss` and at `gopher://eco.utexas.edu/11/mailing`. GAUSS source code may be found at the GAUSS software archive at the American University `http://gurukul.ucc.american.edu/econ/gaussres/GAUSSIDX.HTM`. This page provides many links to other internet pages where code can be found. The developers of GAUSS (Aptech Systems Inc.) provide some support by mail, they can be reached at `info@aptech.com`. Information on libraries and new products can be obtained from their WWW-site at `www.aptech.com`.

In this booklet we give a short introduction to GAUSS. Two things should be kept in mind. First of all, this booklet is no substitute for the complete GAUSS manual but we hope that the reader is able to use the latter more efficiently after reading this one. Many more commands and procedures than the ones discussed in these notes are available. Second, the only way to become proficient in a computer language is by making lots of errors. We recommend doing the exercises and following the examples throughout this book. The setup of this introduction is as follows. In chapter 2 we discuss configuring GAUSS. Data types and operators are discussed in chapter 3 and how to write programs and procedures in chapter 4. We deal briefly with writing GAUSS libraries in chapter 5. File handling is treated in chapter 6. The optional GAUSS module `maxlik` is treated in chapter 7. It is possible to make publication quality graphics in GAUSS, this topic is dealt with in chapter 8. Two more elaborate examples of GAUSS-code can be found in chapter 9. Exercises are given in chapter 10.

Stefan Steinhaus provided useful comments on the first printing. Readers of this introduction are kindly asked to report errors, omissions, and other comments to the author who can be reached by email at `gauss@rhkoning.xs4all.nl`. Sample code can be obtained from the web page `www.rhkoning.com/gauss`.

[2]Note that the email-software that registers subscriptions and cancellations does not accept emails in HTML-format. Hence any such option in the email program of the user trying to subscribe should be disabled when sending mail to the majordomo-software.

Chapter 2

Installing and Configuring GAUSS

In this chapter we discuss installing and configuring GAUSS for Windows. GAUSS runs on 386 (or better) computers with a mathematical coprocessor, using Windows95, Windows98, or Windows NT as operating system. GAUSS requires approximately 7Mb of harddisk space and it will run comfortably on a Pentium computer with at least 32Mb of RAM. Memory management is taken care of by the operating system, but of course, GAUSS will be faster if programs can be executed in RAM memory and disk swapping is avoided. The harddisk requirements increase if the user writes programs and libraries himself, or if one processes large data sets with GAUSS.

GAUSS is shipped on four 3.5" diskettes, each containing one file. The files are called `setup.exe`, `disk1.bnd`, `disk2.bnd`, and `disk3.bnd`. The program is best installed by copying these files into a temporary directory on the harddisk and then by starting the program `setup.exe`. During the installation process the user has to enter the name name of the user and the organization, and the folder where GAUSS is to be installed. The installation program suggests installation in `c:\gauss`. From now on we assume that GAUSS has been installed in that directory. Note that GAUSS does not accept long filenames, so the installation folder should have a directory name of 8 characters at the most. Then one can select a windows group for the GAUSS icons and finally the program is installed. During the installation process a shortcut to the main program is put on the windows desktop.

A second program that needs to be installed is PlayW. This program is needed for conversion of graphs and can be downloaded free of charge from `www.aptech.com`. Again the user has to enter name and organization, and the installation program suggests to install PlayW in `c:\gauss`. The icons will be placed in the same group as the GAUSS icons. After installation the user needs to log off and to log on again so that the conversion program works.

Finally some optional libraries may be installed. These libraries can be bought as add-ons to GAUSS, usually they are not shipped with the main GAUSS program. Usually the diskettes with the library have three directories: `\src`, `\lib`, and `\examples`. The library is installed by copying the files of these directories to the corresponding subdirectories of `c:\gauss`.

The windows helpfile of the main GAUSS program is by default stored in `c:\windows\system32`. This may cause difficulties in accessing the help file from within the GAUSS program. Often, these problems can be solved by moving the helpfile to the directory where GAUSS is installed.

GAUSS itself is started by clicking on the GAUSS-icon that is put on the desktop during the installation. Of course, the program can be started as well by clicking on the file `gauss.exe` in the explorer, or by accessing it from the windows `Start/Programs/Gauss` menu. During startup, the program configures itself according to settings in the file `gauss.cfg` that can be found in the directory where the program is installed. This configuration file can be edited by the user. In this configuration file some variables are set that determine the behaviour of GAUSS. There is no graphical interface to this configuration file. The most important variables that need be set are the path where GAUSS looks for procedures (`src_path`), a variable that determines the smoothness of screen output (`fastio`), and a variable that determines the maximum width of matrices that are printe in the main window (`matwidth`). Another important variable is `max_workspace`. This variable gives the amount of workspace of a GAUSS session in Mb. The value of this variable should be increased if the user encounters 'out of workspace' error messages. Other switches and variables determine paths where data are stored and read and the behaviour of the compilation process. For a new GAUSS user the default values can be retained, except perhaps the four variables mentioned. During startup, GAUSS looks for a program file `startup` (no extension) in the main GAUSS directory. This program file may contain any valid GAUSS commands and these commands are executed during startup. An example of a startup file is

```
/* GAUSS startup file */
 chdir c:\gauss\prog;
 library dutil, maxlik;
```

After executing the startup file, the command window opens and the program gives a command prompt `(gauss)`. The program is now ready to execute commands enetered by the user. The screen immediately after startup is shown in figure 2.1.

GAUSS can be run in command mode or in edit mode. A GAUSS-session is automatically started in command mode, with the command prompt

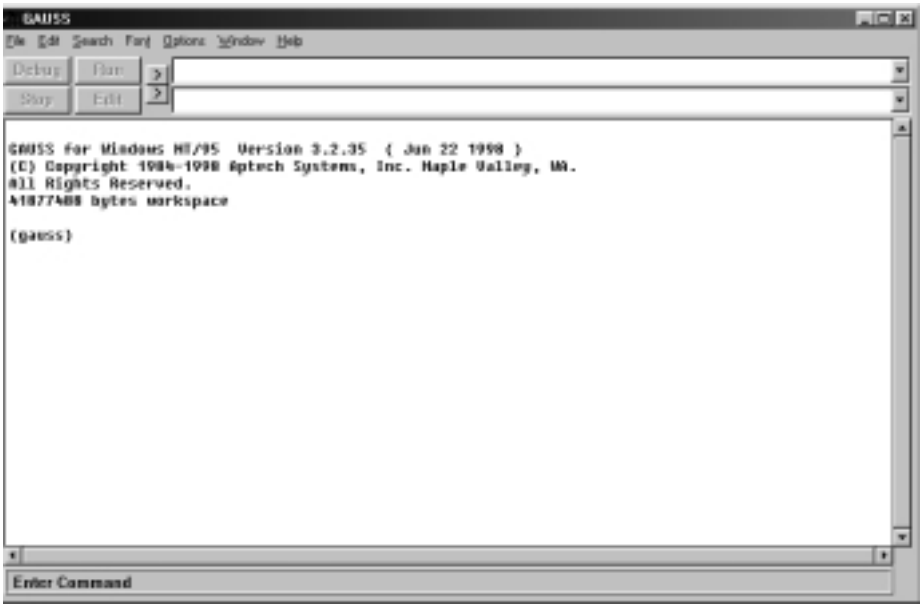


Figure 2.1: GAUSS screen after starting up

`(gauss)`. In command mode, commands are executed immediately after the `<Enter>` key is hit. An example of a simple statement in command mode is

```
>>x=rndn(20,2);y=vcx(x);print y;
```

which is executed by pressing `<Enter>` after the last semicolon. This line consists of three GAUSS commands, separated by semicolons. All GAUSS statements (both in command mode and edit mode) are separated by semicolons. After pressing `<Enter>` the commands are compiled and executed. An example of statements entered in command mode and their output is given in figure 2.2. Even though GAUSS is a windows program, it sometimes behaves differently from most windows programs. There is no functionality attached to the right mouse button. Moreover, it is not possible to put the cursor at a GAUSS statement somewhere in the command window and have it executed by hitting `<Enter>`. It is possible though copy previous GAUSS statements by selecting them using the left mouse button, and copying them to the windows clipboard using `<ctrl><c>`. The statements copied in the clipboard can be pasted at the last command prompt with `<ctrl><v>`.

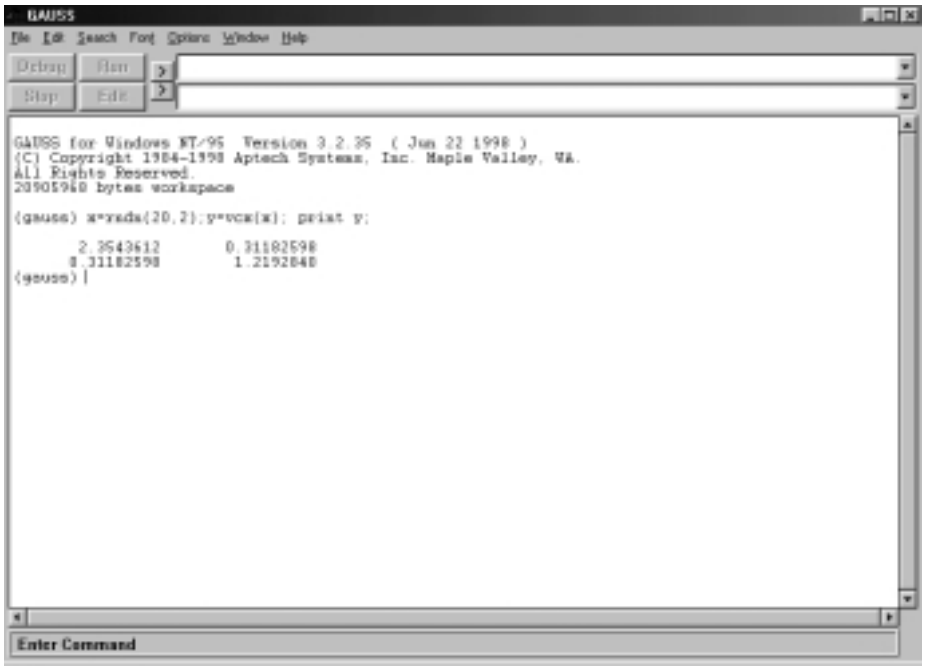


Figure 2.2: GAUSS command window

Only commands at the last GAUSS prompt will be executed if the user hits the `<Enter>` key.

In edit mode, GAUSS commands are typed in a file and they can be stored for later use. The edit mode is started either by entering the command `edit program1.prg` at the GAUSS command prompt in command mode, or by opening the file from the file menu (file/edit). The file is created if it does not exist in the current working directory if one uses the second method. It is not possible to create a file by entering `edit program1.prg` at the command prompt. The current working directory can be changed by the command `chdir c:\gauss\prog\book` in command mode, or by the function `changedir(s)` (with `s` a string with the new directory) in a program. The current working directory is shown by the command `cdir(0)`. The edit window is shown in the top half of figure 2.3.

The edit window has four buttons and two lines on top. The file name (with the complete path) of the file that is being edited is shown in the second line. The first line shows the name of the file that will be run if the

<ctrl><left>	left one word	<ctrl><right>	right one word
<ctrl><home>	begin of text buffer	<ctrl><right>	end of text buffer
<home><home>	beginning of line	<end><end>	end of screen
<F5>	find text	<shift><F5>	find again
<F6>	replace text	<shift><F6>	replace again
<ctrl><g>	goto line	<F1>	on-line help
<F2>	save file	<F3>	run file
<shift><F3>	run current edited file	<F4>	edit file in edit-line
<ctrl><F4>	edit output file		

Table 2.1: Command and edit mode keys

button Run is clicked. The file that is edited is copied to the run-list by clicking the arrow on the left-side of the run-list. The filename that is displayed in the first line is copied to the second line by clicking the arrow on the left-side of the edit-list. All files in the run-list and edit-list are displayed by clicking the arrow-down on the right side of these lines. The file in the edit window is saved by clicking the Save button.

After entering the GAUSS commands as in the top half of figure 2.3, the program can be run. The file name is copied to the run list by clicking the arrow to the left of the run list. The program is run by clicking the Run button, and the output is displayed in the GAUSS command window as in the lower half of figure 2.3. After running this program, the user can either continue by giving interactive GAUSS commands in the command window, or he the user can change the program in the edit window. If one wants to edit the program, the Edit button in the command window can be clicked. This gives focus to the edit window. Of course, the program can be run again by clicking the Run button in either the command window or the edit window.

Some important keystrokes in command- and edit-mode are given in table 2.1. Some of these keystrokes are not applicable when in command mode. When the user hits a key in command mode while running a program, the key that is pressed is handled as interactive input to the program. Online help is available in two different ways. Help can be obtained using the GAUSS help file, but also using the browser. The latter is convenient for searching help on commands that are not standard GAUSS commands, but commands provided by libraries or user defined procedures. The GAUSS help file is opened by Help/Contents or by <F1> in either command or edit mode. The help index is displayed in figure 2.4.

Alternatively, help can be obtained using the browser. Since the user or third parties can add functionality to GAUSS, it is important that the help system can be extended as well. Help on added functionality can be ob-

Figure 2.3: GAUSS edit window (top) and command window (bottom)

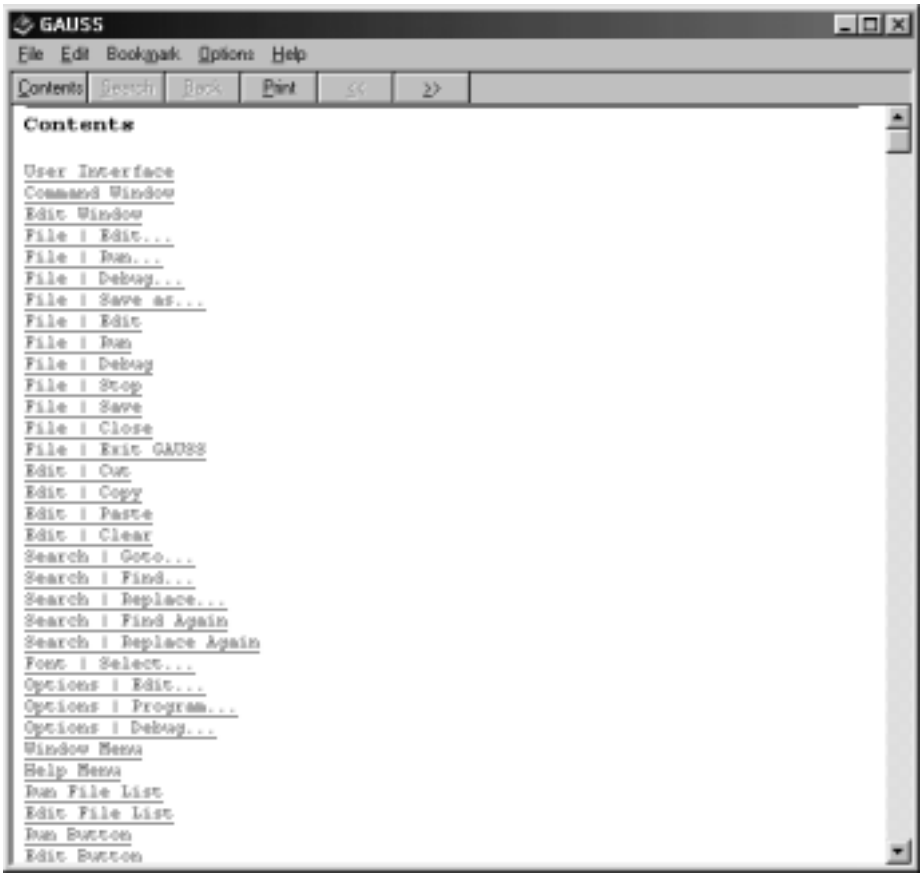


Figure 2.4: GAUSS help file

Figure 2.5: GAUSS help browser

tained using the browser. The browser is started by `File/Browse`. A search topic can be filled in next to `Topic:`, and the browser looks for more information after clicking `Lookup`. Whenever the information on a particular procedure or command is found, the file defining that procedure is opened in the lower panel of the browser window, see figure 2.5. In case the information on the topic is located in the GAUSS help file, the appropriate page is opened. A more detailed decription on the browser can be found in the browser helpfile that is distributed with GAUSS. The file is called `browshlp.txt` and is located in `c:\gauss`. By default the browser will load this file when it is started. This can be disabled by unchecking the option `Load Browser Help on Startup`. This option and some others can be set under `Options/View` in the browser window.

Commands entered during an interactive GAUSS session are stored in the file `command.log` and errors are stored in `gauss.err`. A GAUSS session

is ended by `File/Exit GAUSS`. All temporary files are deleted and the run and edit lists are cleared.

Chapter 3

Data Types and Operators

GAUSS knows only two data types and it is not possible to have new data types defined by the user. The two data types are matrices and strings. Matrices are two-dimensional arrays. The most simple matrix is a scalar which is a 1×1 matrix. In general, a matrix has n rows and m columns and is referred to as an $n \times m$ matrix. The elements of a matrix are either numbers (numbers are stored in double precision, so that the number of significant digits is 15 or 16, numbers must be in the range $4.19E-307 <| x |< 1.67E308$) or characters. In the latter case the elements may consist of up to eight characters so that the amount of memory required for each element does not exceed 8 bytes. GAUSS does not have the special data type integer. A matrix may consist of elements of both types, for example a data matrix may have characters with variable names in the first row and the remaining rows can contain the data. In the virtual memory version of GAUSS there is no limit on the size of the matrix apart from the amount of workspace available. The other data type is the string data type. Again, there is no limit to the length of the string, apart from the workspace available.

In a GAUSS program or in a GAUSS-session there is no need to declare variables before they are used[1]. On the other hand, local variables in a procedure must be declared before they are used using the `local` statement, see section 4.2.

A matrix can be initialized in five ways. First, it can be initialized using the `let`-statement as in `let x={ 1 2 3, 4 5 6}` or in `let x[2,3]= 1 2 3 4 5 6` where the matrices are filled row-wise. Second, a matrix can be initialized by concatenation of existing matrices, for example[2]

[1] This is no longer the case if the autoloader and autodelete state are set to values different from their default values. See the manual for details.

[2] Each of these commands can be entered consecutively in the GAUSS command window.

```
let a={1 2, 3 4};
let b={5 6};
let c={10, 11, 12};
x=a|b;
x=x~c;
```

Here, a is a 2×2-matrix, b is a 2-row vector and c is a column vector of length 3. Data are read row-wise and consecutive rows are separated by a comma. The matrix x is initialized by vertical concatenation using the |-operator and then the vector c is added by horizontal concatenation with the ~-operator. It is possible to initialize an empty matrix by `let x={}` and to concatenate matrices to this matrix x: `x=x~a`. After this initialization, x equals a. The third way of initializing a matrix is by using cetrain special matrix functions like `x=ones(n,k)`, `x=zeros(n,k)`, etc. The fourth way of initializing a matrix is by keyboard input: `x=con(2,3)`. After this command, the user is asked to enter the elements of the matrix interactively. The last way to initialize a matrix is by reading the matrix from disk if that matrix has been saved from a previous GAUSS session[3].

A matrix can be printed on screen or to another output device (like a file or printer) by typing its name: `print x` followed by <enter> (or the line `print x;` in a GAUSS program). If we type the command `print x;` after the last line of the example above in the GAUSS command window, we obtain:

```
print x;

       1.0000000       2.0000000       10.000000
       3.0000000       4.0000000       11.000000
       5.0000000       6.0000000       12.000000
```

In fact, the same result would be obtained if we would not type the command `print`. The value of a variable is printed by `x;`. However, `print x;` is better readable code than `x;`.

A matrix with characters can be initialized by explicit initialization as in `a={"x1","x2"}` or, equivalently, `a="x1"|"x2"`, by concatenation, or by reading from disk. In order to print the vector a, it must be preceded by a $-sign as in $a.

Strings can be initialized analogously to matrices. A string may be initialized by explicit assignment, as in `s="this is a string"`. One can concatenate strings as in:

[3]For more details on file input/output we refer to chapter 6.

```
string1="this is a string";
string2="and this a second string";
string3=string1$+string2;
```

Analogously to keyboard initialization one can initialize a string with keyboard using the command `string=cons`. Note that it is not necessary to specify the (single) dimension of the string. The appropriate amount of memory to store the string will be allocated during the input from the keyboard. Strings saved to disk in a previous GAUSS session are retrieved by `loads s=string`. The contents of the file `string.fst` are read into `s`. A string can be printed by typing its name (without a $-sign preceding it): `string3` followed by `<enter>` will print `this is a string and this is a second string`.

Certain character combinations are not allowed in strings. In general, the backslash \ indicates an escape character. Sometimes a backslash is needed in a string (for example to indicate a path). In that case, one should use a double backslash as for example in `path="c:\\gauss"`. Some other special characters are `\b` (backspace), `\e` (escape), `\f` (formfeed), `\g` (beep) and `\t` (tab). `\123` generates a character whose ASCII-value is '123'.

Sometimes one wants to use the value of a string in an expression where a literal is expected. For example, if GAUSS encounters a command to open a file, GAUSS must be able to make a distinction whether the string passed is the actual filename (a literal) or a variable that contains the filename. Functions associated with file input/output interpret strings usually as literals. One must use the caret (^, also known as the substitution operator) in order to get a filename from a string variable. Consider the following example:

```
data="dataset";
load x1=data;
load x2=^data;
```

The matrix `x1` will contain the data stored in a file with the name `data` and the matrix `x2` will contain the data stored in the file `dataset`. In the first case, `data` is interpreted as a literal and in the second case it is substituted by the contents of the string.

Strings, character matrices and numbers can be converted into each other. An example may be useful. Let `c` be a vector with character elements. An element of `c` can be transformed into a string by concatenating it with "". Similarly, a string can be transformed into a character element by preceding it with 0. See the following examples:

```
c="x1"|"x2";            /* character vector */
```

```
s=""$+c[1];              /* string */
z="string";              /* string */
b=0$+z;                  /* character vector with one
                            element */
```

It is mandatory that "" and 0 come first in the concatenation. "" denotes an empty string and 0 denotes an empty character vector. A numerical value can be transformed in a character value using the function `ftocv` and into a string using `ftos`. Conversely, a string can be transformed into a numerical variable by the function `stof`. The following lines create a character vector with character elements "var1" to "var10":

```
n=seqa(1,1,10);       /* n is a 10-vector with 1,2,...,10 */
var_n=0$+"var"$+ftocv(n,1,0);
```

The second argument of the function `ftocv` is the minimum field width (1 in this case, as the shortest number is represented by 1 character) and the last element indicates the number of decimal places.

GAUSS does not perform any type checking on the variables, so it is possible to run the following program without encountering any errors:

```
s="this is a string";
s=s+5;
```

GAUSS distinguishes between 'regular' and element-by-element operators that are defined on both datatypes. Later, it will be shown that the latter operations can be very convenient. The more important regular operators defined on matrices are + (addition), - (subtraction), * (matrix multiplication), / (division, `x=b/A` is the solution to `Ax=b`), % (modulo division) and ! (factorial). All these operators are defined on matrices of appropriate dimensions only. Element-by-element operators are performed elementwise. Examples are `.*` (element-by-element multiplication), `./` (element-by-element division), `.^` (element-by-element exponentiation, the same as ^) and `.*.` (Kronecker product). Element-by-element operators are obtained by preceding the regular operator with a dot `.`. Other important matrix operators are ' (transpose), ~ (horizontal concatenation) and | (vertical concatenation). Two important operators defined on strings and character matrices are $+ (string concatenation) and ^ (string variable substitution).

To illustrate the difference between 'usual' matrix multiplication and elementwise multiplication consider this example:

```
x=1~2;
y=3|4;
```

```
x
   1.0000000        2.0000000
y
   3.0000000
   4.0000000
x*y
   11.000000
x.*y
   3.0000000        6.0000000
   4.0000000        8.0000000
```

Of course, matrices need to be conformable when elementwise operators are used. Two matrices x and y are elementwise conformable if:

- they are of the same size, the operations are carried out on corresponding elements;
- x is a scalar and y is a matrix, the scalar is operated with every element in the matrix;
- x is a column vector of length n and y is an $n \times m$ matrix, the vector is swept accross the matrix;
- x is a row vector of length m and y is an $n \times m$ matrix, the vector is swept down the matrix;
- x is a row vector and y is a column vector, the result will be the outproduct of both vectors.

Various relational operators are defined in GAUSS. Most operators can appear in two forms other than their 'regular' form: the element-by-element form and the `$` form for comparisons between character data and between strings. The element-by-element form is obtained from the regular form by preceding that form with a dot `.`. The relational operators available are `==` (is equal to), `<` (is less than), `<=` (is less than or equal to), `>` (is greater than), `>=` (is greater than or equal to) and `/=` (is unequal to). These expressions will evaluate to 1 (true) or 0 (false) if they are used in their regular forms. If the relational operator is used in its element-by-element form it evaluates to a matrix of 0's and 1's, depending on whether the condition holds for that pair of elements. If `x` and `y` are matrices of the same dimensions with floating point numbers, `x==y` will evaluate to 1 or 0 (depending on whether *all* elements of `x` and `y` are equal or not) while `x.==y` will evaluate to a matrix (of the same dimensions as `x` and `y`) with 1's and 0's, depending on which elements are equal or not. Consider for example

```
x=1~2;
y=1~3;
x==y
   0.0000000
x.==y
   1.0000000         0.0000000
```

Finally, the following logical operators are implemented: AND, OR, NOT, XOR (exclusive OR) and EQV. Again, these operators can be used element-by-element by preceding them with a dot ..

Another example may illustrate the use of an elementwise operator. The following program simulates data for a probit model:

```
nobs=100;          /* number of observations */
true_beta=0|1|-1; /* vector with parameters */
x=ones(nobs,1)~2*rndn(nobs,2); /* x matrix with
                     intercept and regressors */
y=(x*true_beta+rndn(nobs,1)).>0; /* vector with 0's
                     and 1's */
```

In the last line, every element of y is set to 0 or 1 depending on whether the corresponding element of the vector x*true_beta + rndn(nobs,1) is smaller than 0 or not. This code is much faster and certainly more transparent than 'usual' code with some kind of loop. Elementwise operators should be used as much as possible since this kind of code is executed much faster than similar code that performs the same task for each element individually. This is illustrated by the program on page 22 where elementwise operators and a loop are compared.

Sometimes one needs a submatrix of a given matrix. The desired rows and columns can be selected by indicating them between square brackets: x[.,1] selects the first column of a matrix x, x[1:4,.] selects the first four rows of x. The upper left 4×4 block of x is selected by x[1:4,1:4]. It is also possible to extract a submatrix using a vector with indices as in

```
c=1|4|6;
a=x[c,.];
```

which selects rows 1, 4, and 6 into a new matrix a. Of course, the maximal element of c should not exceed the number of rows of x and all elements of c should be positive integer numbers.

A matrix may consist of both character and numerical data, as in the example

```
x=30|40;
```

```
y="Pete"|"Joe";
z=y~x;
```

If one prints the matrix z using `print z` the first column will display very small numbers, not the character contents of y. As discussed earlier, character data are printed correctly if the matrix is preceded by a $, so `$z[.,1]` (the first column of the matrix z in the example above) will give sensible results. Both the character and numerical content of z are printed correctly using the GAUSS function `printfmt`, as in `printfmt(z,0~1)`. The first argument of `printfmt` is the matrix to be printed. The second argument is a row vector with entries 0 and 1, depending on whether the corresponding column of z contains character data (0) or numerical data (1).

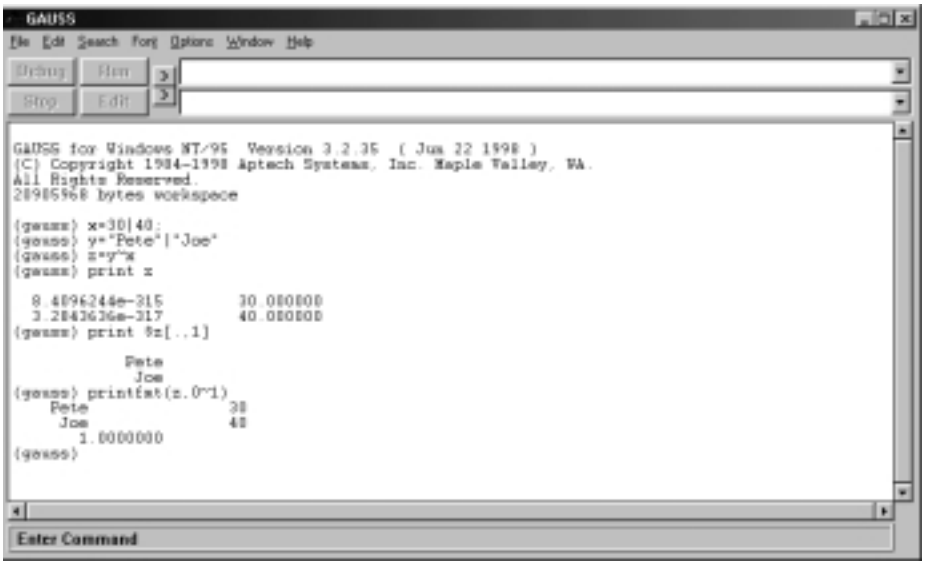


Figure 3.1: A matrix with both numerical and character elements

Chapter 4

GAUSS Programs and Procedures

4.1 GAUSS Programs

GAUSS programs can loosely be defined as files with valid GAUSS commands. A program file may consist of both the actual code of the program and additional procedures specific to that program. Comments in a GAUSS program or procedure are opened by `/*` or `@` and closed by `*/` or `@` respectively. Variable and procedure names in GAUSS are not case sensitive.

As a first command of a GAUSS program, the user can start with `new;` which clears the workspace. All matrices and procedures from previous programs are deleted from memory. A program can be ended with the `end;` statement which closes all open files and terminates the program. The `pause(10);` statement halts execution of the program for 10 seconds and the `system;` statement exits GAUSS.

An important element in any program is the flow control. Various keywords are available in GAUSS to determine whether a piece of code should be repeated some times or whether it should be executed at all. A GAUSS-loop is started using the `do-while` statement and ended by the `endo` statement. Within the loop, it is possible to jump to the top of the loop with the `continue`-statement and to break out of the loop with the `break`-statement. In that case the program proceeds with the first command following `endo`. Consider the following example:

```
i=1;
do while (i<=100);
 i=i+1;
 print "i=" i;
endo;
```

Note that a Boolean expression `i<=100` determines whether the code within the loop should be executed again or not. Alternatively, the loop can be

controlled using the `do-until`-statement as in

```
i=1;
do until (i>100);
 i=i+1;
 print "i=" i;
endo;
```

It is not advisable to perform matrix operations using a `do-while` loop if they can be performed using elementwise operators instead. The following program generates 100000 random numbers uniformly distributed between 0 and 10 and classifies them into the intervals $[0, 3)$, $[3, 7)$, and $[7, 10]$. (`hsec` is a standard GAUSS function that returns time elapsed since midnight in hundredths of a second.)

```
r=100000;
v=10*rndu(r,1);

et1=hsec;
v1=zeros(r,1);
i=1;
do while (i<=r);
 if (v[i]<3);
  v1[i]=1;
 elseif (v[i]>7);
  v1[i]=3;
 else;
  v1[i]=2;
 endif;
 i=i+1;
endo;
et1=(hsec-et1)/100;

et2=hsec;
v2=(v.<3) + 2*(v.>=3).*(v.<=7) + 3*(v.>7);
et2=(hsec-et2)/100;

et3=hsec;
v3=dummy(v,3|7);
v4=v3[.,1] + 2*v3[.,2] + 3*v3[.,3];
et3=(hsec-et3)/100;

print "loop " et1;
print "vectorized " et2;
print "dummy " et3;
```

```
print "ratio loop/vectorized " (et1/et2);
end;
```

Classification is much faster if done using the elementwise operators, in this particular example the 'vectorized'-code is almost three times as fast. In fact, even faster code is

```
v2=1+(v.>=3)+(v.>=7);
```

Code is executed conditionally using the `if-endif`-statements. Within an `if-endif`-branch one can use `elseif` and `else`-statements for further conditioning. Both the `if` and `elseif`-statements must be followed by a scalar expression which determines whether the code should be executed or not. Each `if`-statement must be ended with an `endif`-statement. Consider the following example (% is the modulo division-operator):

```
i=1;
do while (i<=20);
 if (i%2==0);
  print "i is even " i;
 elseif (i%3==0);
  print "i is odd and divisible by 3" i;
 else;
  print "i is odd and not divisible by 3" i;
 endif;
 i=i+1;
endo;
```

After writing a program in the GAUSS-editor, the program is run by clicking on the arrow left of the run-list and clicking the `Run` button. If the program is already on top of the run-list clicking the `Run` button suffices and the program is saved automatically before compilation and execution. In the DOS-version of GAUSS the user could use the debugger to debug a program. The debugger is not yet implemented in the Windows version, despite the presence of a button `Debug`. If the program has syntactical errors, GAUSS will issue error warnings during the compilation. Suppose for example that the period after `i=i+1` is omitted, then compiling the program results in

```
(gauss) run c:\gauss\prog\book\if-exam.prg
I=I+1 ENDO
        ^
C:\GAUSS\PROG\BOOK\IF-EXAM.PRG(10) : error G0008 : 'ENDO' :
Syntax error
2 error(s)
```

A GAUSS program need not be run by clicking the Run button. Alternatively it can be run from the GAUSS-command prompt using the command run, so for instance the command run c:\gauss\prog\book\if-exam.prg compiles and executes the program if-exam.prg stored in the subdirectory c:\gauss\prog\book. If no explicit path is given, GAUSS tries to find the program file in the current directory first, and then searches along the path in the environment variable src_path set in the configuration file gauss.cfg. During execution a program can be stopped by clicking the Stop button in the command window.

4.2 GAUSS Procedures

Procedures are the building blocks of GAUSS. Many useful procedures are provided with the installation and other handy procedures are shipped with the GAUSS modules or with commercial extensions to GAUSS. GAUSS derives its flexibility from the possibilities for users to write their own procedures. These procedures can be very simple or complex, even though it is recommended that complex procedures be broken into in a few simpler ones. In this section we will discuss writing a procedure and some important standard procedures that belong to the standard installation of GAUSS. Many procedures of interest to econometricians can be found in the GAUSS software archive, see page 2.

A procedure is created along the following steps. First, the source code must be written in a file[1]. That file must have the same name as the procedure[2], and have extension .g. For example, the code of the procedure boxcox must be in the source file boxcox.g. This requirement implies that a procedure name can have up to eight characters. The file with the source code must be placed in a subdirectory that is listed in the path for program files (the variable src_path in gauss.cfg, see chapter 2).

The actual code for the procedure consists of the following five parts:

1. procedure declaration,

2. declaration of the local variables,

3. actual code of the procedure,

[1] If the procedure is specific to one program (for example, a procedure that calculates a specific likelihood function to be optimized) the code of the procedure may also be placed in the file that contains the code of the GAUSS program. In this case, that procedure can not be called from other programs.

[2] If the procedure is part of a library, this is not necessary. Creation of libraries is discussed in chapter 5.

4. returning values,

5. end of the procedure.

It is good practice to document the most important features of the procedure in the first couple of lines, between the comment terminators `/*` and `*/`. If the user needs help on the procedure he can use the help browser to obtain help. The help browser puts the source code of the procedure in the browser window, hence it is convenient if documentation is in the top of the source file.

Every procedure starts with a declaration, as for example `proc (1) = boxcox(x,l);`. The number of returns is put between parentheses, if only one object is returned this can be left out as in `proc boxcox(x,l);`. The parameters of the procedure are passed after the procedure name. In this case, there are two parameters: `x` and `l`.

The second element of the procedure is the declaration of the local variables. All variables in GAUSS are global ones, unless they are preceded by the keyword `local` when they are declared. All global variables are accessible from within the procedure and are not declared in one way or another. Local variables are declared as in `local z;`. After this command, `z` can be initialized, it is not initialized by its declaration. A local variable may have the same name as a global variable, within the procedure where it has been declared as the local variable temporarily 'overrides' the global variable of the same name. All parameters are passed by reference, that is, no copy to a new local variable is made. Hence, global variables can be changed from within procedures.

Part 3 of any procedure is generally the most interesting part. In this part the actual calculations are performed. In this section, other procedures may be called.

The fourth element of the procedure is returning the result of the calculations. The number of elements returned must coincide with the number of returns given in the declaration. An example of the return statement is `retp( z );`. If the procedure has no returns, this statement can be skipped, if more than one element is returned, the elements are separated by comma's as in `retp( z, x );`. Finally, every procedure is terminated with the `endp;` command.

An example of a procedure that calculates the Box-Cox transform (the Box-Cox transform is a transformation in statistics that transforms a variable according to $x^{(\lambda)} = \frac{x^{\lambda}-1}{\lambda}$ if $\lambda \neq 0$ and $x^{(\lambda)} = \ln x$ if $\lambda = 0$) is

```
proc (1)=boxcox1(x,l);
 local z;
```

```
 if (l==0);
  z=ln(x);
 else;
  z=(x.^l-1)/l;
 endif;
 retp(z);
endp;
```

An improved version of this procedure is listed below. In that procedure, first it is checked whether the second argument (l) is a scalar. If this is not the case, an error message is printed using the `errorlog`-command and the procedure returns value -1. After this check the actual calculations are performed, depending on the value of the second parameter. This procedure could be extended with some help comments on top.

```
/* boxcox2, procedure to calculate the Box Cox transform
input: x: n x k matrix with positive elements
       l: scalar, parameter of Box Cox transform
output: y: n x k matrix with Box Cox transformations of
           each element of x;
*/
proc boxcox2(x,l);
 if (rows(l)/=1 or cols(l)/=1);
  errorlog "boxcox2.g: l must be a scalar";
  retp(-1);
 endif;
 if (x>0);
  if (l==0);
   retp( ln(x) );
  else;
   retp( (x.^l-1)/l );
  endif;
 else;
  errorlog "boxcox2.g: x must be positive";
  retp(-1);
 endif;
endp;
```

This version uses less workspace because no local variables are declared. Moreover, note that it is possible to exit from the procedure with a `retp`-statement from anywhere within the procedure. The procedures checks whether the parameters are admissable: the first parameter should be a matrix with positive numbers and the second parameter should be a scalar. If the second parameter is not a scalar, the procedure prints an error (both to the screen of the command window and to the error log file `gauss.err`)

and returns -1. If the second parameter is a scalar, the procedure checks whether the first parameter is a matrix with positive numbers. If this is the case, the actual calculations are performed, otherwise another error message is printed.

Procedures can be called in different ways (`eigrs2` is a procedure that calculates the eigenvalues and eigenvectors of a real, symmetric matrix):

```
x=boxcox(y,0.5);
boxcox(y,0.5);
{va,ve}=eigrs2(h);
call eigrs2(h);
```

In the first case, the result of the procedure is stored in the matrix `x`. This matrix `x` is created and initialized automatically, if necessary. In the second case, the output of the procedure `boxcox` is copied to screen. The output of the procedure `eigrs2` consists of two elements. The first matrix in the `retp`-statement of `eigrs2` is stored in `va` and the second matrix in that statement in `ve`. In the fourth case, the procedure is executed and all output is discarded. This way of calling a procedure may be useful if the return indicates successful completion of the procedure only (as is usually the case with, for example, the procedure `xy`) and one is not interested in the result.

One should note that GAUSS is not a very 'safe' language in the sense that it hardly performs any type checking at either compilation or run time. Hence, the user should do this when the procedure is written. Sometimes a procedure uses global variables (for example the `ols`- and `dstat`-procedures to be discussed below). In general, it is better to pass global variables as parameters to the procedure. Global variables are variables that exists throughout a GAUSS program, whereas local variables are created when the procedure that defines them is called. Passing global variables as parameters makes the procedure better usable in another context: global variables tend to be there when you need them, but also when you don't expect them. The following program is valid GAUSS code; it shows the dangers of accessing global variables from within a procedure.

```
new;
a=3;
call change_a();
print a;

proc (0)=change_a();
 a=5;
endp;
```

The printed result is 5 even though most users would expect the result 3 from the `print a`-statement.

A special kind of argument of a procedure is a pointer to another procedure. This is useful when writing some general purpose routines that take a procedure as their argument. For instance, the derivative of a procedure can be approximated by

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

with h a small number. This general purpose procedure to approximate a derivative can be programmed taking a pointer to the function f as its argument. See the example below:

```
new;
x=1|2|3;
numerical_derivative(&x2,x,1e-8);
end;

proc numerical_derivative(&f,x,h);
 local f: proc;
 retp( (f(x+h)-f(x-h))/(2*h) );
endp;

proc x2(x);
 retp( x^2 );
endp;
```

Pointers to these procedures are obtained by preceding the name of the procedure with an ampersand (`&`), a convention well-known to C- and C++-programmers. Note that the symbol for the procedure to be passed in the argument list of the procedure `numerical_derivative` (`f` in this case) must be declared as a local procedure in the local declaration list (`local f:proc`). It is the responsibility of the user that `f` is called correctly within `numerical_derivative`. For instance, if the procedure `x2` were redefined as

```
proc x2(x,k);
 retp( x^k );
endp;
```

GAUSS would give an error message because the call `f(x+h)` in the procedure `numerical_derivative` would be invalid, as this procedure has one parameter only.

GAUSS is shipped with many standard procedures. A complete list can be obtained from the help file. A list with procedures to generate random

rndbeta	$B(a,b)$ random variates
rndgam	$\Gamma(\alpha,1)$ random variates
rndn	$N(0,1)$ random variates
rndnb	NegBin(p,k) random variates
rndp	$P(\lambda)$ random variates
rndu	$U(0,1)$ random variates

Table 4.1: Random numbers

cdfbeta	Beta distribution function
cdfbvn (*)	bivariate normal distribution function
cdfbvn2 (*)	bivariate normal distribution function
cdfbvn2e	bivariate normal distribution function
cdfchic	complement χ^2 distribution function
cdfchii	inverse χ^2 distribution function
cdfchinc	non-central χ^2 distribution function
cdffc	complement F distribution function
cdffnc	non-central F distribution function
cdfgam	incomplete Γ function
cdfmvn (*)	multivariate normal distribution function
cdfn (*)	standard normal distribution function
cdfnc(*)	complement standard normal distribution function
cdfni	inverse standard normal distribution function
cdftc	complement Student's t-distribution
cdftci	inverse complement Student's t-distribution
cdftnc	non-central Student's t-distribution
cdftvn	trivariate normal distribution function
pdfn	standard normal density function

Table 4.2: Distribution functions (those functions marked with a * are also available in logarithmic form)

numbers is given in table 4.1 and a list with procedures to calculate statistical distribution functions is given in table 4.2. The arguments to call these procedures vary, so they are omitted in these tables.

Two important GAUSS procedures are the procedure to perform linear regression and the procedure to calculate descriptive statistics. Both procedures use both local and global variables. First we discuss the regression procedure. The procedure estimates parameters in the linear model

$$y_i = \beta' x_i + \varepsilon_i, \qquad i = 1,\ldots,N. \tag{4.1}$$

In equation (4.1), the k-vector x_i is the vector with regressors that may or may not include a constant term. The disturbances ε_i are assumed to be independently distributed with zero mean and (constant) variance σ^2. The parameters of the model (β and σ^2) can be estimated by

```
{vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat}
```

```
 =ols(dataset,depvar,indvar);
```

The parameters of this procedure are `dataset` (a string variable containing the name of the dataset), `depvar` (a character vector with one element or a scalar pointing to the row in the dataset with the dependent variable) and `indvar` (a character vector with the names of the independent variables or a vector with the row indices of the independent variables). If `dataset` is a null string, the actual vector of the dependent variable and the matrix with independent variables are assumed to be passed to the procedure as `depvar` and `indvar`. The following two calls yield the same results:

```
call ols("testols,"Y1","X1"|"X2);
call ols(0,y,x1~x2);
```

assuming that the dataset `testols` contains the same values for the variables as the vectors `y`, `x1`, and `x2`. The way OLS estimates are calculated is partly determined by global variables. These global variables are also known as 'flags'. The first is `__con` (`con` is preceded by *two* underscores) if this variable is set to 0 the regression is estimated without an intercept. The default value is 1 so that an intercept is included. A second global variable is `__miss`, this one determines how missing values are treated. The default value is 0 so that the procedure assumes all observations are valid. A third important global variable is `_olsres` (`olsres` is preceded by one underscore). If this variable is set to 1, the Durbin-Watson test statistic for autocorrelation is calculated and the OLS-residuals are determined. The default value is 0. Finally, the variable `__altnam` may be set to a character vector with the names of the variables, with the name of the dependent variable as the last element. Detailed information on this procedure can be obtained from the online help by `<alt>-h` followed by `h` and `ols`.

Another useful standard procedure is `dstat`. This procedure calculates the mean, standard deviation, minimum, maximum and number of valid cases of a dataset or a datamatrix. Its syntax is

```
{vnam,mean,var,std,min,max,valid,missing}
 =dstat(dataset,vars);
```

Again, `dataset` is a string variable with the name of the dataset to be analyzed and `vars` may be either a character vector or an index vector. If `vars` is 0, descriptive statistics of all variables in the dataset are listed. If `dataset` is 0, `vars` is assumed to be a datamatrix that is analyzed. Treatment of missing values is determined by the global variable `__miss`. This variable has default value 0 so no checking for missing values is performed.

An example where both the `ols`- and `dstat`-procedure are used is

```
new;
nobs=100;
x=ones(nobs,1)~3*rndn(nobs,3);
beta=0|1|-1|0.5;
sigma=1.5;
y=x*beta+sigma*rndn(nobs,1);

_olsres=1;
__altnam="CONSTANR"|"X1"|"X2"|"X3"|"DEPVAR";
call dstat(0,x~y);
print;print; /* two empty lines */
call ols(0,y,x);
end;
```

The output of this program is shown in figure 4.1.

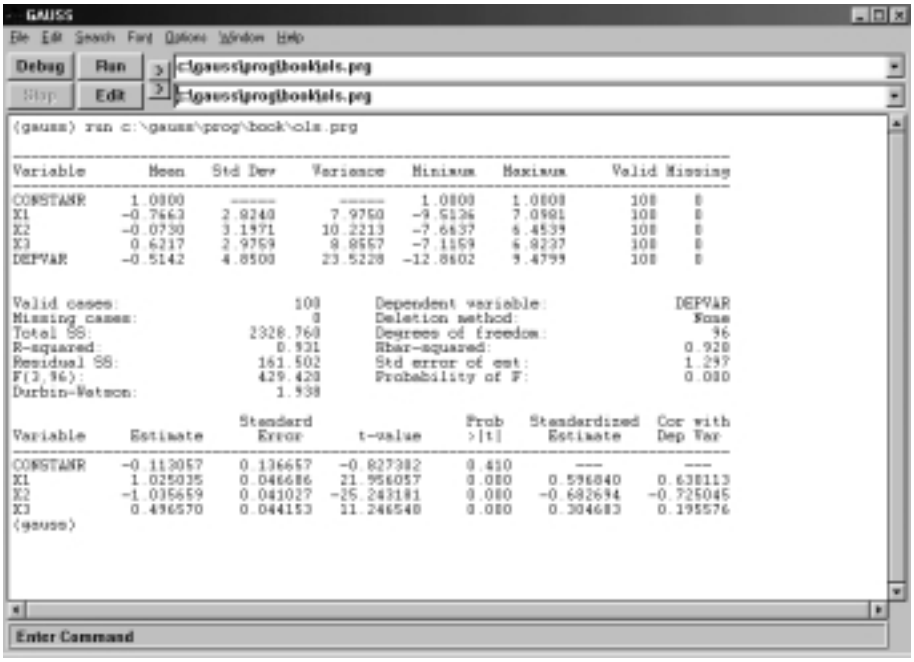


Figure 4.1: `dstat` and `ols` output

Chapter 5

Libraries

GAUSS is a modular language. It can be extended by libraries that add new procedures to the language. These libraries are distributed both as commercial add-ons to GAUSS as well as free software. In this section, we will describe how add-on libraries can be used as well as how one can write a library.

Before we discuss writing libraries, we need to discuss the way GAUSS searches for unknown references (like a matrix or a procedure). Suppose GAUSS is in the process of compiling a program and in encounters a reference to a procedure that has not yet been compiled. The way GAUSS searches for the unknown object using the autoloader the autoloader. First, the autoloader searches in the current directory (the current directory can be retrieved in command mode by the command `cdir(0)`) and then along the searchpath given by `src_path` in the file `gaussi.cfg` (see also section 2). The exact way how is searched for unknown references is determined by the state of the autoloader and the autodelete state. Suppose GAUSS encounters this line of code:

```
{b,s2}=ols_estimate(y,x);
```

Since `ols_estimate` is not an intrinsic GAUSS function, code for this function needs to be located and compiled. If the autoloader is turned off, then the procedure must have been declared before with the command

```
external proc ols_estimate;
```

otherwise and 'undefined symbol' error is given. If the autoloader is turned off forward references (ie, references to objects not already defined) are not allowed. If both the autoloader is on and the autodelete-state is on GAUSS searches for the unknown object along the following paths. First, GAUSS tries to find it in the user library, then in user-specified libraries, then in the

GAUSS library and finally it searches for files with a `.g` extension in the current directory and along the path listed in `src_path`. In case the autoloader is on but the autodelete-state is off, GAUSS does not search for files with a `.g` extension. Moreover, forward references to objects not listed in the libraries are not allowed in that case. Compilation time is longest when both the autoloader and the autodelete-state are on, but that situation is most convenient to the user. In the remainder of this section we assume that both the autoloader and the autodelete-state are on. Under `Options/Program` one finds a menu with checkboxes. By checking or unchecking these boxes the user can change the state of the autoloader and the autodelete state. Moreover, by unchecking the GAUSS or user library one can prevent that these libraries are searched when the autoloader looks for an unknown object. Considering the current speed of computers, it is recommended to leave all checkboxes checked.

A GAUSS library is best thought of as an index file where the program can find the exact location of references. Libraries are activated by a command like `library maxlik, bstat;`. After activation, procedures and matrices defined in these libraries become available in a program. Two libraries are activated by default when GAUSS is started: the `gauss`- and `user`-libraries (unless this feature is turned off under `Options/Program`). The GAUSS-library is a library with native GAUSS procedures like `dstat`, `bstat` and many others. The `user`-library is empty at the moment GAUSS is installed on ones system, but the user can add own procedures to this library (see below).

GAUSS libraries are stored as ASCII-files with extension `.lcg` in the directory specified by the variable `lib_path` in `gaussi.cfg`. An example of such a library file is `course.lcg`:

```
c:\gauss\prog\book\testlib3.src
    ml_estimation                   : proc
    ols_estimation                  : proc
```

This library consists of two objects: two procedures (`ml_estimation` and `ols_estimation`). If the `course1`-library has been activated by the command `library course1` and the compiler encounters a reference to an object in this library, GAUSS 'knows' where to find the code for that particular object and the file containing that object (in this example `testlib1.src`) will be compiled. In fact, all code in `testlib1.src` will be compiled so it is sensible not to create too large library source code files. It is not necessary to have all the code for one particular library in one file, as the following example shows:

```
/*
** tscs.lcg - Time Series/Cross Sectional Analysis Library
```

```
** (C) Copyright 1988-1996 by Aptech Systems, Inc.
** All Rights Reserved.
*/

tscs.dec
    _ts_ver                         : matrix
    _tsmodel                        : matrix
    _tsstnd                         : matrix
    _tsmeth                         : matrix
    _tsise                          : matrix
    _tsmnsfn                        : string
    _ts_mn                          : string

tscs.src
    tscs                            : proc
    _tsgrpmeans                     : proc
    _tsprtp                         : proc
    tscsset                         : proc
    _tsfile                         : proc
```

This library is the library file of the time-series/cross-section library, one of the libraries commercially available. A list with all commercially available libraries can be found in Appendix B. All code for this library is found in two files: `tscs.dec` (a file with declarations of global variables, see below) and `tscs.src` (a file with the actual procedures that make up the library). Usually library files with source code have extension `.src` or `.arc`. A second file with source code (say, `testlib2.src`) is added to our `course1`-library by

```
lib course1 testlib2.src
```

at the GAUSS-command prompt.

The list of active libraries and the directory where these library files are stored can be found by giving the command `library` without any library names in the command window. Most libraries use global variables to allow the user to determine how calculations are made. These global variables can be declared and initialized at compile time using the `declare`-statement. In the following source code file, the OLS-estimator is calculated, and an estimate for the variance of the error term in the linear regression model.

```
#include testlib3.dec

proc (0)=testlibset;
 _df_correction=1;
```

```
endp;

proc (2)=ols_estimation(y,x);
 local n,k,sxx,sxy,b,e,s2;
 n=rows(x);
 k=cols(x);
 sxx=x'x/n;
 sxy=x'y/n;
 b=inv(sxx)*sxy;
 e=y-x*b;
 if (_df_correction==1);
  s2=e'e/(n-k);
 else;
  s2=e'e/n;
 endif;
 retp( b, s2 );
endp;
```

When this file is compiled, the `include`-statement 'inserts' the contents of the file `testlib3. dec` into `testlib3.src` and the resulting code is compiled. The file `testlib3.dec` contains the line

```
declare _df_correction?=0;
```

Here, the global variable `_df_correction` is declared and initialized. When the code of the procedure `ols_estimation` is compiled, GAUSS 'knows' that `_df_correction` is a global variable with value `0`, unless the user has initialized this variable earlier. In that case, `_df_correction` is *not* reinitialized, because it is declared using `?=`. If the usual `=` were used, `_df_correction` would be reinitialized, even if it were initialized before. For details of initializing variables we refer to the manual.

When a library is activated, help on its procedures can be obtained in the browser window. Open the browser window by `File/Browser` and type the name of the procedure in the lookup window. Clicking the `Lookup` button puts the file with the source code of that procedure in the browser window. Note that the top of the file is displayed in the window, hence it is good practice to put comments in the top of the file.

CHAPTER 6

FILE INPUT/OUTPUT

GAUSS stores data in two different (binary) formats. It saves data as a matrix or as a string (the files have extension `.fmt` and `.fst` respectively), or as a GAUSS data set. In the the case that it is stored as a GAUSS data set , the data are saved in two files, one file contains the actual data and the other file contains header information (like the names of the variables stored in the data set). These files have extensions `.dat` and `.dht`. The advantage of storing data in a GAUSS data set is that some procedures in GAUSS can access that data set directly. If all the variables are not needed in the analysis, it may be easier to not read all data into memory but to only read the variables that are needed.

Exchanging data with other programs used to be a bit difficult in GAUSS. The DOS-version of GAUSS supported only import of files that were in some sort of ASCII-format. Data were converted from this format to GAUSS using a utility program that is shipped with GAUSS. This program and the data conversion are discussed in Appendix A. The Windows version, on the other hand, support import from and export to a variety of binary formats of other programs. Data stored in these formats can be read into memory using `import` and they can be converted to a GAUSS-data set using `importf`. We discuss reading data into matrices first, and then we discuss these two procedures.

An ASCII-file with numeric data is read into a vector `x` with the `load`-command: `load x[]=file.asc`. The data are read row by row from the file `file.asc` and must be separated by a white space (ie, a space, tab or return). The data in `file.asc` must be data suitable for storage in the GAUSS data type matrix: they must be numerical or a sequence of characters starting with the character `'a'...'z'` or their uppercase equivalents. When GAUSS encounters an invalid first character (for instance &), it reads that and consecutive characters incorrectly. Suppose for instance that `file.asc` contains the following data:

```
(gauss) load x[2,2]=file.asc
(gauss) print x

       1.0000000        2.0000000
       1.0000000        2.0000000
(gauss)
```

Figure 6.1: Reading an ASCII-file with invalid characters

```
1 2
& 7
```

Reading these data with `load x[2,2]=file.asc` yields the result shown in figure 6.1. Suppose on the other hand that the dimension of the data set to be read in is not known. Reading data as in `load x[]=file.asc` reads all data row-wise into a column vector `x`. If needed, the vector `x` can be reshaped into a matrix by `y=reshape(x,100,5)` with `y` having 100 rows and 5 columns. Of course, the column vector `x` should have 500 elements in this case.

Data can also be read using the `import`-statement. The data may be in ASCII-format or in some supported binary format. A list with all the supported binary formats is given in the left two columns of table 6.1. The `import`-procedure has three arguments and two returns:

```
{x,nm}=import("file.dat",r,s)
```

The first argument is a string with the name of the file to be read. The type of the data in the file is determined by the extension of the file, see table 6.1. If necessary, this can be overridden by setting the global variable `_dxftype`

import		export	
extension	format	extension	format
`wks wk1 wk2`	Lotus v1–v2	`wks`	Lotus v1.0
`wk3 wk4 wk5`	Lotus v3–v5	`xls`	Excel v2.1
`xls`	Excel v2.1-v7.0	`wq1`	Quattro v1.0
`wq1 wq2 wb1`	Quattro v1-v6	`wrk`	Symphony v1.0
`wrk`	Symphony v1.0-v1.1	`db2`	dBase II
`db2`	dBase II	`dbf`	dBase III
`dbf`	dBase III/IV, Foxpro, Clipper	`db`	Paradox v3.0
`db`	Paradox	`csv txt asc`	ASCII character delimited
`csv txt asc`	ASCII character delimited	`prn`	ASCII formatted
`prn`	ASCII packed	`dat`	GAUSS data set
`dat`	GAUSS data set		

Table 6.1: Supported import and export formats

to an appropriate value (for instance, `_dxftype="xls"`). The second argument is a string, giving either the range of the spreadsheet to be read (as in `r=A1..G3"`), or as a format descriptor for a packed ASCII-file (see the online help for more information on this descriptor). If this argument is set to 0, all data are read in. The third argument is the sheet number of the page to be read in. Usually this is 1. When `import` reads spreadsheets it assumes that the first row has the names of the variables. If this is not the case, the global variable `_dxwkshdr` should be reset to 0. The output `x` contains the actual data and the vector `nm` has the names of the variables. In order to read the data, GAUSS writes them to a buffer first. The size of this buffer is 1 Mb, if the data set is bigger than that the global variable `_dxbuffer` (the size of the buffer in Mb's) can be increased.

The `import` statement reads both numerical and character variables. If we read the data set `file.asc` with the `&` in the second line, the first column is interpreted as a column with character values and the second column is a numerical column, as can be seen in figure 6.2 Instead of reading a data set

```
(gauss) {x,nm}=import("file.asc",0,1)

*******************************************************************************
                          GAUSS Data Import Facility
*******************************************************************************

Begin import...
Import completed
(gauss) x

  2.4209217e-322        2.0000000
  1.8774495e-322        7.0000000
(gauss) %x

              1
              &
(gauss)
```

Figure 6.2: Reading an ASCII-file with invalid characters

into GAUSS memory, it is also possible to convert an existing data set into a GAUSS data set. Instead of the `import`-command the `importf`-command is used. An example is

```
y=importf("test.xls","test",0,1)
```

where the Excel-spreadsheet `test.xls` is converted to a GAUSS data set with the same name. The procedure returns either 1 (successful conversion) or 0 (unsuccessful conversion).

Compiled procedures, strings, and matrices can be saved in a binary format. The advantage of saving a procedure in binary format is that it does not have to be recompiled again. Usually though, one wants to save a matrix or a string (array) so that these can be used later for further analysis. The

general format of the save-command is save name=symbol where name is a literal or a referenced string and symbol is a symbol (a matrix or a string, for example). Examples are (x is a matrix and s is a string):

```
save x;
save names=s;
save c:\tmp\xx=x;
save path=c:\tmp x;
```

In the first case, x is saved into the binary file x.fmt in the current directory, in the second case, s is saved into the binary file names.fst in the current directory, in the third case, x is saved into the file xx.fmt in the directory c:\tmp and in the final case x is saved into the file x.fmt in the directory c:\tmp and all further objects saved with the save-command will be placed in this directory, unless an explicit filename is given as in the third example. The extension .fmt indicates a matrix written to disk and the extension .fst indicates a string written to disk.

A matrix file created using the save-command can be loaded into the workspace using the load-command, for example by load x (x.fmt is loaded from the current directory into x) or load x=c:\tmp\xx (in this case c:\tmp\xx.fmt is loaded into x). A binary file containing a string can be loaded with the loads-command, which works analogously to the load-command. Note that the load-command is used both to read ASCII-data into a GAUSS matrix and to read matrix files saved earlier in a GAUSS-session. It is not possible to read ASCII-data using the loads-statement.

It is not always practical to store data in a matrix, sometimes it is more convenient to store them in a data set on disk, especially if the data set is large. A data set can be created in two different ways: from within GAUSS or by using the exportf-command. First, one can use the saved-command as in saved(x,data set,vnames) where x is the matrix to be saved in the datafile, data set is a string variable containing the name of the data set and vnames is a vector with the names of the columns of x. Upon successful completion, saved returns the value 1. It is customary to assign variable names in capitals for numerical variables and in lowercase for character variables. This convention determines whether GAUSS handles character data and numerical data correctly by default. If no vector with variable names is passed (*i.e.*, vnames is set to 0), GAUSS creates a vector with variable names automatically, with names X1, X2, etc.

```
/* reading and writing datasets */

x=3*rndn(1000,3);
y=rndn(1000,1);
c=(y.<0).*"ltzero" + (y.>=0).*"gtzero";
```

```
data=x~c;

/* uppercase-> numerical variables
   lowercase-> character variable
*/
vnames="X1"|"X2"|"X3"|"sign";
file_name="testdata";
saved(data,file_name,vnames);

z=loadd(file_name);
```

Data can be read into memory using the `loadd`-command as in the example above. This, however, is possible for small data sets only. If the data set is large data can be read by reading successive chunks from the datafile. Alternatively, data can be read row-wise from a data set, as in the example below.

```
/* reading data row-wise */

file_name="testdata";
vnames=getname(file_name);
i=1;
let data={};
open fh=^file_name;
number_rows=rowsf(fh);
do while not(eof(fh));
 data_row=readr(fh,1);
 data=data|data_row;
endo;
close(fh);
```

First, one assigns a file handle to the file that is going to be read. Then the data set is accessed using the `readr(fh,1)`-command. The first argument is the file handle, the second argument is the number of rows that is going to be read. Finally, the data set is closed. All open files in a GAUSS session may be closed with `closeall`. The variable names of the columns in the data set can be retrieved by the command `vnames=getnames(data set)` where `data set` is a string variable with the name of the data set. A similar approach can be used to write rows to a file using a `writer`-command.

Alternatively, a data set is created by the `export`- or `exportf`-command. In the first case a matrix `x` and a vector with variable names `nm` are written to a file as in `y = export(x,"data.dat",nm)`. The extension of the filename determines again the format of the data to be saved, see table 6.1. Moreover, a GAUSS data set can be exported in the same way using the `exportf`-command as in `y = exportf("data.dat", "data.xls", nm)`.

Note the extension of the first argument. It is mandatory even though only GAUSS data sets are converted this way.

Output of programs that is shown in the command window can be saved in a file as well. All output printed in the command window is copied to a file by the following command: `output file=program.out reset`. The file `program.out` is created in the current directory. GAUSS stops copying output to this file after either the command `output off` or an `end` statement in a program. Note the qualifies `reset`. This qualifier ensures that the file `program.out` is created if necessary. Should a file with that name already exist, it is overwritten. The command `output file=program.out on` *appends* output to the file `program.out` if it exists and it creates the file if it does not exist.

CHAPTER 7

MAXIMUM LIKELIHOOD ESTIMATION

A very convenient GAUSS library is the maximum likelihood library[1]. This library contains procedures to maximize a loglikelihood function, to perform bootstrap analysis of nonlinear models, and to perform profile likelihood analysis. Note that this library is not distributed with the GAUSS-program itself: it must be bought separately[2].

Computer language code for global optimization of functions is widely available nowadays (see for instance Press, Teukolsky, Vetterling, and Flannery (1992)), but most of this code works well only with very well behaved functions. The optimization routines in the maxlik-library are based on Dennis and Schnabel (1983) and they are designed to deal with likelihood functions that are not globally concave. The maxlik-library has many flags that can be set by the user, here we will only deal with the most important ones. The reader is advised to read the Maximum Likelihood Manual for more information. Of course, on-line help is available as soon as the maxlik-library has been activated.

We will discuss the use of the maxlik-library by means of a simple example. Consider the normal linear regression model with the data generated according to

$$y_i = \beta' x_i + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, \sigma^2). \tag{7.1}$$

The loglikelihood function of this model (apart from a constant) is given by

$$\ell(\beta, \sigma) = -\frac{N}{2} \ln 2\pi - \frac{N}{2} \ln \sigma^2 - \frac{1}{2\sigma^2} \sum_i (y_i - \beta' x_i)^2 \tag{7.2}$$

[1] In this manual we deal with version 4.0.20 of the maxlik-library. Previous version of this library have a similar structure and the main procedures are used similarly.

[2] A related third-party library is the cml-library to do constrained maximum likelihood estimation.

The maximum likelihood estimates of β and σ are the values $\hat{\beta}$ and $\hat{\sigma}$ that maximize function (7.2). In the following example we activate the maxlik-library, generate data according to model (7.1) and the maximum likelihood estimates of the vector β and the scalar σ are determined using the maxlik-procedure:

```
library maxlik;
maxset;

output file=maxlik1.out reset;
nobs=5000;
beta=1|-1|0.5|-0.25;
s=2;
x=ones(nobs,1)~3*rndn(nobs,3);
y=x*beta+s*rndn(nobs,1);
theta0=ones(5,1);

/* set global variables */
_max_ParNames="BETA1"|"BETA2"|"BETA3"|"BETA4"|"S";
_max_Algorithm=2;
_max_LineSearch=1;
_max_CovPar=3;

{x,f,g,cov,retcode}=maxlik(x~y,0,&loglik,theta0);
call maxprt(x,f,g,cov,retcode);

end;

proc loglik(theta,z);
 local y,x,b,s;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 retp( -0.5*ln(s^2)-0.5*(y-x*b)^2/s^2 );
endp;
```

Apart from a call to the maxlik-procedure, the user only has to program a function that has a column vector with contributions to the loglikelihood as output. The maxlik-procedure takes a pointer to this likelihood function as one of its arguments and the function is maximized (in fact, the negative of the loglikelihood is minimized) using numerical derivatives and default optimization algorithms.

The central procedure in the maxlik-library is the maxlik-procedure. This procedure has four arguments and its output consists of 5 elements:

```
{x,f,g,cov,retcode}=maxlik(dataset,vars,&loglik,theta0);
{x,f,g,cov,retcode}=maxlik(data,0,&loglik,theta0);
```

In the first line, the data are read from the file `dataset` and the vector `vars` is a character vector with the names of the variables used in the analysis or a numerical vector with the indices of the selected variables. In the second case, the data are passed as a matrix `data` (see also the example above). The third argument is a pointer to a procedure that returns a vector with contributions to the loglikelihood function. This procedure has two arguments: the first is the parameter vector and the second is the data matrix. If the data are read from file, the order of the columns in the data matrix is specified in the vector `vars`. The final argument of `maxlik`-procedure is `theta0`, a vector with starting values for the optimization.

In the example above the parameter vector of the `loglik`-procedure `theta` equals β and σ stacked on top of each other. Each row of the data matrix `z` consists of an observation $(\, x_i' \quad y_i \,)$ and correspondingly, each element of the output vector with contributions to the loglikelihood is equal to

$$-\frac{1}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}(y_i - \beta' x_i)^2.$$

The first output vector, `x`, is the vector with estimated parameters. The scalar `f` is the value of the function at the minimum (minus the mean loglikelihood), the vector `g` is the value of the gradient evaluated at `x`, the matrix `cov` is the covariance matrix of the parameters and the scalar `retcode` indicates whether the optimization has terminated normally (in which case `retcode = 0`) or not. These results can be printed using the `maxprt`-procedure:

```
call maxprt(x,f,g,cov,retcode);
```

Optimization of the loglikelihood function proceeds in two steps. At each iteration a direction d is determined along which line a 'better' value for $\hat{\theta}$ is searched, and a step length λ is calculated which improves the value of the likelihood function. The new value of $\hat{\theta}$ is then determined as $\hat{\theta}_{i+1} = \hat{\theta}_i + \lambda_i d_i$.

The direction is commonly computed as (omitting the subscript i indexing the iteration) $d = Ag$, with A some square matrix and g the gradient of the loglikelihood function. Different choices of A are available, they correspond to different optimization algorithms. The optimization algorithm is set by the global variable `_max_Algorithm`. A first choice of A is to set $A = I$, with I the identity matrix. This method is the steepest descent algorithm (`_max_Algorithm=1`). Another choice is to set A equal to the inverse

of the Hessian of the loglikelihood function, which is known as the Newton-Raphson algorithm (_max_Algorithm=4). This is method is rather computer intensive (at every iteration the matrix of second derivatives must be calculated and inverted) and works only well if the loglikelihood function is well-behaved. So-called secant methods approximate the Hessian of the loglikelihood function by adding updates to an Cholesky decomposition of the Hessian. This Cholesky decomposition is then used to determine a direction d. The approximation of the Hessian improves with the number of iterations. Two such secant methods are implemented: the Broyden, Fletcher, Goldfarb, and Shanno algorithm (_max_Algorithm=2) and the Davidson, Fletcher and Powell algorithm (_max_Algorithm=3). Another well-known optimization algorithm in econometrics is implemented: the Berndt, Hall, Hall and Hausman method (_max_Algorithm=5). This method estimates the Hessian by the average of the outer products of the gradients of each contribution to the loglikelihood function. The default optimization algorithm as set in maxset is the BFGS-algorithm (_max_Algorithm=2) and for most problems there is no need to use another optimization algorithm as it is reasonably robust against scaling and conditioning of the model. Of course, this default (as any other default) can be changed by editing the file maxlik.dec where these flags are initialized.

Given the direction d, one has to determine the step length. The function to be minimized by λ is $\ell(\theta + \lambda d)$. The way λ is calculated is determined by the global variable _max_LineSearch. The simplest method is to set $\lambda = 1$ (_max_LineSearch=1). A better procedure is to fit a quadratic or cubic function (in λ) to $\ell(\cdot)$ and then to choose λ such that this approximating function is maximized (_max_LineSearch=2). Other methods of obtaining a step length are based on a sequence of values of λ (_max_LineSearch=3, 4, 5). The default choice is _max_LineSearch=2.

Optimization of the loglikelihood function stops if the algorithm has converged, or if the maximum number of iterations is reached. The latter is set by _max_MaxIters and the default maximum number of iterations is 10000. Convergence is reached when the tolerance of the gradient (which is defined as a constant times the largest element of the gradient vector) is below is below $1e-4$. The tolerance level can be set with the global _max_GradTol.

During the optimization process, intermediate results are printed at the screen. This can be surpressed by setting the global variable __output to 0. Resetting it to 1 prints output to the screen. Sometimes one wants to inspect the values of the coefficients, gradient, of Hessian during the optimization, especially if convergence problems are encountered. If the global variable _max_Diagnostic is set to 2 or 3 intermediate results are stored.

Suppose the optimization is halted because of an error (or by <c>). Current values of the parameters, function, gradient, Hessian, and steplength are all stored in a data buffer _max_Diagnostic. Because this buffer is a hybrid vector with both character elements and numerical elements, data can only be retrieved using vread. A list of names of the information stored in _max_Diagnostic is obtained by vlist(_max_Diagnostic). The value of, say, the parameters is now retrieved as in

```
x=vread(_max_Diagnostic,"params")
```

The covariance matrix of the parameters can be estimated in three different ways. The first estimate is the inverse of the Hessian (_max_CovPar =1), the second estimate is the inverse of the matrix of cross-products of first derivatives (_max_CovPar=2) and the third estimate is an estimate of the covariance matrix that is consistent even if the data are heteroscedastic (see White (1982), _max_CovPar =3). The latter estimate can also be used for testing the specification of the model.

By default, the maxlik-procedure uses numerical derivatives during the optimization. Of course, this may result in many evaluations of the loglikelihood function at each iteration. It is possible to supply a procedure that calculates the derivative of the loglikelihood function analytically. That procedure must have two arguments: a vector with parameters and a matrix with the data. The output of this analytical gradient procedure is a matrix with as many rows as the data matrix and the columns contain the derivatives with respect to the parameters. An analytical gradient is used by the maxlik-procedure if the global variable _max_GradProc is set to a pointer to that gradient procedure.

The derivatives of the loglikelihood function in the normal linear model are given by

$$\frac{\partial \ell_i}{\partial \beta} = \frac{1}{\sigma^2}(y_i - \beta' x_i)x_i$$
$$\frac{\partial \ell_i}{\partial \sigma} = -\frac{1}{\sigma} + \frac{1}{\sigma^3}(y_i - \beta' x_i)^2$$

where ℓ_i denotes the contributions of the ith observation to the loglikelihood. This analytical derivative is programmed as the procedure loglikgd below. Each row of the matrix returned from this procedure contains the derivatives given in the two equations above.

```
library maxlik;
maxset;
```

```
output file=maxlik2.out reset;

nobs=5000;
beta=1|-1|0.5|-0.25;
s=2;
x=ones(nobs,1)~3*rndn(nobs,3);
y=x*beta+s*rndn(nobs,1);
theta0=ones(5,1);

_max_GradProc=&loglikgd;
_max_GradCheckTol=1e-3;
_max_Diagnostic=2;
/* now optimization using analytical gradient */
{x0,f,g,cov,retcode}=maxlik(x~y,0,&loglik,theta0);
call maxprt(x0,f,g,cov,retcode);

end;

proc loglik(theta,z);
 local y,x,b,s;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 retp( -0.5*ln(s^2)-0.5*(y-x*b)^2/s^2 );
endp;

proc loglikgd(theta,z);
 local y,x,b,s,e,gb,gs;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 e=y-x*b;
 gb=e.*x/s^2;
 gs=-1/s+e^2/s^3;
 retp( gb~gs );
endp;
```

The global variable _max_GradTolCheck is the maximal difference that is allowed between the numerical gradient and the analytical gradient. If the analytical gradient differs by too much from the numerical gradient, maxlik terminates with an error and prints both gradients. This can be used to debug the computer code of the analytical gradient. For example, consider

the Tobit model. In this model, data are generated according to:

$$y_i^* = \beta' x_i + \epsilon_i, \qquad \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$
$$y_i = \max(0, y_i).$$

Only y_i and x_i are observed so that the loglikelihood function up to a constant), is

$$\ell(\beta, \sigma) = \sum_{y_i=0} \ln \Phi(-\beta' x_i/\sigma) + \sum_{y_i>0} \left[-\frac{1}{2} \ln \sigma^2 - \frac{1}{2\sigma^2} (y_i - \beta' x_i)^2 \right].$$

and the gradient is

$$\frac{\partial \ell}{\partial \beta} = - \sum_{y_i=0} \frac{\phi(-\beta' x_i/\sigma)}{\sigma \Phi(-\beta' x_i/\sigma)} x_i + \sum_{y_i>0} \frac{1}{2\sigma^2} (y_i - \beta' x_i) x_i$$
$$\frac{\partial \ell}{\partial \sigma} = \sum_{y_i=0} \frac{\phi(-\beta' x_i/\sigma)}{\sigma^2 \Phi(-\beta' x_i/\sigma)} x_i + \sum_{y_i>0} \left[-\frac{1}{\sigma} + \frac{1}{\sigma^3} (y_i - \beta' x_i)^2 \right]$$

For example, in the code below data are generated according to the Tobit model, and the model is then estimated.

```
/* tobit model with error in gradient */

library maxlik;
maxset;

output file=maxlik3.out reset;

nobs=500;
beta=1|-1|0.5|-0.25;
x=ones(nobs,1)~3*rndn(nobs,3);
y=x*beta+2*rndn(nobs,1);
y=(y.>0).*y;


_max_GradProc=&loglikgd;
_max_GradCheckTol=1e-3;

{x0,f,g,cov,retcode}=maxlik(x~y,0,&loglik,ones(5,1));
call maxprt(x0,f,g,cov,retcode);

end;

proc loglik(theta,z);
```

```
 local y,x,b,s;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 retp( (y.==0).*ln(cdfn(-x*b/s)) +
  (y./=0).*(-0.5*ln(s^2)-0.5*(y-x*b)^2/s^2) );
endp;

proc loglikgd(theta,z);
 local y,x,b,s,e,gb1,gs1,gb0,gs0,a;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 e=y-x*b;
 a=-x*b/s;
 gb0=-(pdfn(a)./cdfn(a)).*x/s;
 gs0=(pdfn(a)./cdfn(a)).*a/s;
 gb1=e.*x/s^2;
 gs1=-1/s+e^2/s^3;
 retp( ((y.==0).*gb0+(y./=0).*gb1)~((y.==0).*gs0+(y./=0).*gs1) );
endp;
```

However, there is an error in the gradient: the 10th line of the gradient should read

```
 gs0=-(pdfn(a)./cdfn(a)).*a/s;
```

GAUSS checks whether the numerical gradient and the analytical gradient give the same results. In this case they differ, and an error message is printed, see figure 7.1. From the printed output it is clear that the error is in the gradient for σ only, because the other elements of the gradient vector coincide. The error is solved by adding a minus sign in front of the expression for `gs0`.

As a final check of the computer code, it is strongly recommended that the likelihood function and/or the gradient are tested using simulated data. This need not be time consuming and can prevent problems later on.

During the optimization process, it is possible to change the global variables that determine the way the optimization routine looks for 'better' parameter values, and to change the current point where the loglikelihood is evaluated. By pressing <h> during the optimization process, a screen appears where the user can select what global variables need to be changed.

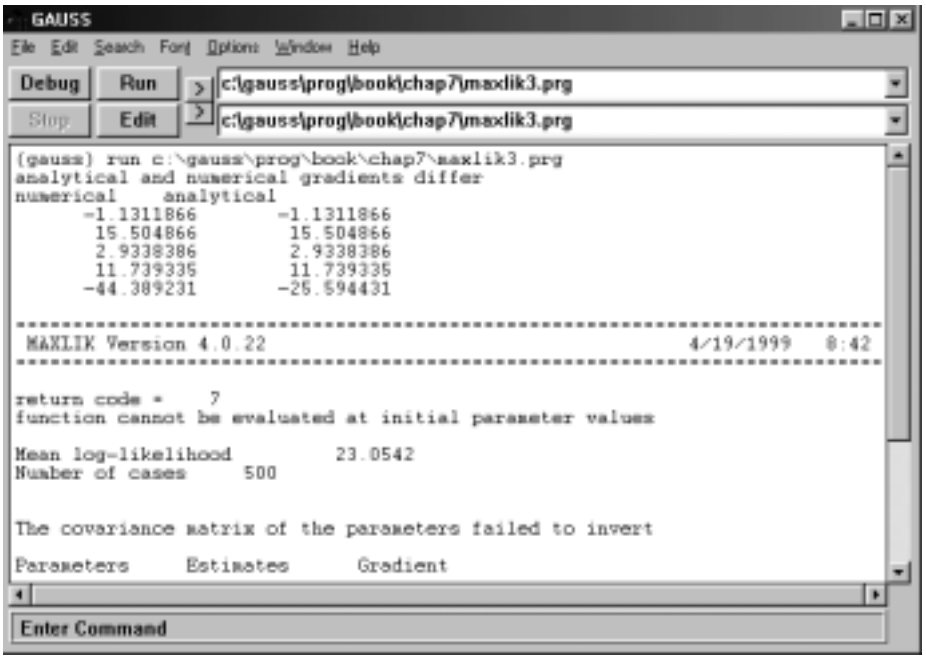


Figure 7.1: Error in gradient

Note that it is suggested to press <alt><e> to edit the parameter vector. This keystroke is not applicable in the windows version of GAUSS, one should press <e> instead.

In this chapter we have discussed only a few procedures of the maximum likelihood library. The library has more procedures than the ones discussed here. It has a procedure `maxdensity` to calculate kernel density estimates, a procedure `maxprofile` to calculate profile likelihood traces, a procedure `maxboot` to perform bootstrap simulation of parameters, and a few others. The user is referred to the manual or the online help for more information.

Chapter 8

Graphics

GAUSS is equipped with graphics capabilities. With a few commands one is able to plot two- and three dimensional data and the graphs can be saved for incorporation in a text document. First we discuss some elements of the graphics library in GAUSS. Then we discuss how graphs can be included in wordprocessors such as Microsoft Word and LaTeX.

8.1 Creating Graphics in GAUSS

In this section, we discuss configuration of the graphics library and the most important commands. The graphics library of GAUSS is configured analogously to the way GAUSS is configured by editing the file `gauss.cfg`, but not all settings in this file are relevant to the Windows version of GAUSS. For instance, the printer and the print layout can be set from the menubar of the command window, following `File/Printer` or `File/Print Layout...`. It is important that the directory that contains the file `gauss.exe` is listed in the DOS search path. It will not be possible to run the graphics program if that is not the case. It is possible to create multiple windows with each window containing a graph. This feature will not be discussed here. General help on the graphics library is obtained by asking help `@pqg` in the online help program.

The graphics library is activated by `library pgraph` and global variables are set to their default values by calling the procedure `graphset` (a procedure without parameters). GAUSS graphics are created in four steps:

1. Activation of the graphics library.
2. Reading and formatting of the data.
3. Setting global variables to special values.
4. Calling the actual graphics procedure needed.

xy(x,y)	logx(x,y)	logy(x,y)	loglog(x,y)
bar(v,h)	hist(x,v)	histf(x,f)	histp(x,p)
box(g,h)	xyz(x,y,z)	surface(x,y,z)	contour(x,y,z)

Table 8.1: Graph types

In this chapter we focus on the third and fourth step. After a graph is displayed on screen the user has two options. Pressing <ESC> lets the program continue and pressing <enter> yields a menu with some saving and printing options.

In table 8.1 we give the commands that generate graphics. The names of the procedures indicate the type of graph it creates. The most important graphs that can be generated are two- and three-dimensional plots (xy and xyz), contour and surface plots of three-dimensional data (contour and surface), and histograms (hist, histf, and histp).

To illustrate the use of the graphics library, we give a simple example first. A graph of the standard normal density function is generated in the following program:

```
/* density function of standard normal */

library pgraph;
graphset;

grid_size=0.02;
x=seqa(-3,grid_size,6/grid_size);
y=pdfn(x);

xy(x,y);
```

The standard normal density function is evaluated on a grid of the horizontal axis; the distance between successive points is 0.05. If the distance would be too large, say 1.0, the graph would be too jagged. After running this program a new window appears with the output of the program. The window is shown in figure 8.1.

Output is partly determined by global variables of the graphics library. Of course, these values must be set before the graph is drawn using one of the commands in table 8.1. Global variables are set by calling a procedure (for example, title("example plot")) or by explicit assignment (for example, _pdate=0). Not all global variables are applicable to each graph type. Global variables are reset to their default values by calling the procedure graphset. Axis can be named with xlabel("text on x-axis"),

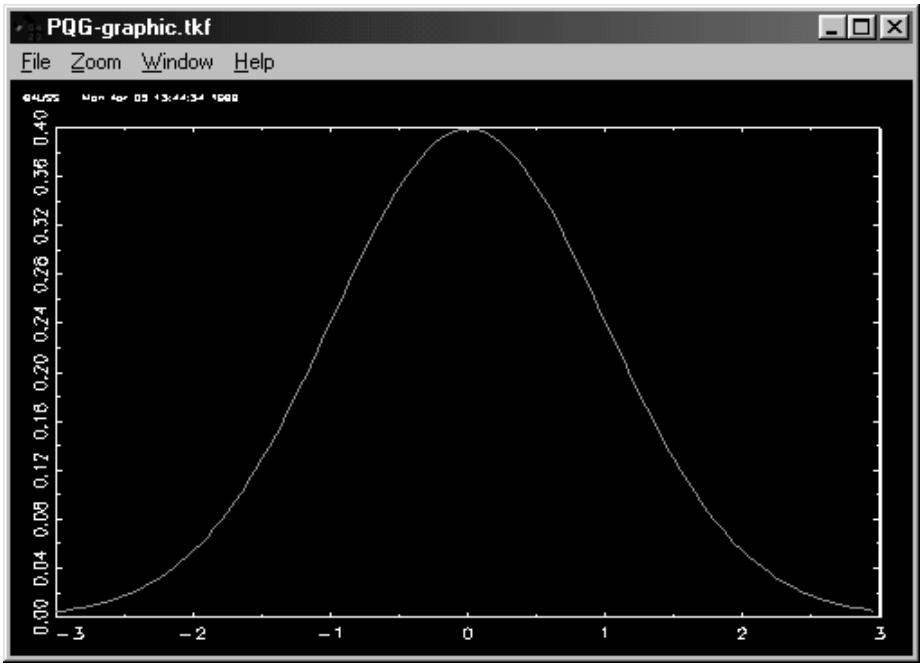


Figure 8.1: Output of `density.prg`

`ylabel("text on y-axis")`, and `zlabel("text on z-axis")` and a title is given with `title("title of the plot")`. The date and time of creation of the graph is printed in the upper left corner. This feature can be turned off by setting the global variable `_pdate=0;`. If `_pdate` is set to a string, then this string will be printed in the upper left corner and the date will be appended.

The commands `hist(x,v)`, `histf(f,c)` and `histp(x,v)` all produce histograms. They differ only in type of information displayed on the vertical axis (absolute numbers, frequencies or percentages) and input requirements (raw data and a vector of breakpoints (`hist` and `histp`) or frequencies and a vector with category labels (`histf`). In the file below, we generate a vector with normal distributed numbers and a histogram is generated using $-1.5, -1, 0, 1$ and 1.5 as breakpoints so that the first bar corresponds to observations $x \leq -1.5$, the second category to $-1.5 < x \leq 1$, etc.

```
library pgraph;
graphset;
```

```
x=rndn(100,1);
b=-1.5|-1|0|1|1.5;
title("histogram of normal distribution");
ylabel("frequency");
xlabel("x value");
_pdate=0;
call hist(x,b);
```

Two dimensional graphs can be plotted using the `xy(x,y)`xy command. Of course, the arguments `x` and `y` must be of equal length unless one wants to index the observations. The command `xy(1,y)` plots the curve $(1, y_1)$, $(2, y_2)$, etc. The second argument need not be a vector: if a matrix (with the same number of rows as `x`) is passed as an argument, each column will be graphed. Consider for example the following program

```
library pgraph;
graphset;

ngrid=100;
x=seqa(-3,6/ngrid,ngrid);
y1=pdfn(x);
y2=1./(pi*(1+x^2));

_plegctl=1~4~1~0.32;
_plegstr="Gaussian density\000Cauchy density";
xlabel("ordinate");
ylabel("density function");

xy(x,y1~y2);
```

In this example, two density functions are graphed, viz. the density of the normal distribution and the density function of the Cauchy distribution. The second argument passed to the `xy`-procedure is a 100×2-matrix and each column is graphed as a separate line. The legend in the graph is set by two variables. The 4-row vector `_plegctl` determines where the legend is placed. The first element determines whether the coordinates are set in plot coordinates (1), inches (2), or pixels (3), the second element gives the font size of the legend (between 1 and 9) and the final two elements give the horizontal and vertical coordinate of the lower left corner of the legend (in units specified by the first argument). In our example, coordinates are given in plot coordinates, font size is 4 and the coordinate of the lower left corner is $(1, 0.32)$. The text of the legend is specified in `_plegstr`. Different curve labels are separated by the ASCII-0 character `\000`. The label

of the first curve is 'Gaussian density' and the label of the second curve is 'Cauchy density'. If the number of labels given in _plegstr is less than the number of columns in y empty labels will be printed in the legend otherwise the order of the curves in the legend correspond to the order of the columns in y. Additional text messages can be plot in the graph using the global variables _pmsgctl and _pmsgstr. By its default settings set using the graphset procedure, a line is drawn through the points plotted. The type of line plotted is controlled by the global variable _plctrl. If this variable is set to -1, only individual points are plotted and these points are not joined by a line. The symbol at each point is controlled using _pstype.

It is also possible to draw three dimensional curves and contour lines are graphed using the surface and contour procedures. Both procedures have three arguments: a *row*-vector x of length k, a vector y of length p and a $p \times k$ matrix z. The rows of z correspond to the elements of y and the columns of z correspond to the elements of z. A graph of the surface above the (x, y) plane is drawn by surface(x,y,z) and a two dimensional graph with isocontour curves is drawn by the command contour(x,y,z). A three dimensional graph of a function is obtained with the xyz(x,y,z) procedure. This procedure yields 3D results analogous to the xy procedure discussed above. As an example, we give the following program that generates three 3D plots based on a bivariate normal density function:

```
library pgraph;
graphset;

ngrid=51;
x=seqa(-3,6/ngrid,ngrid);
y=x;
x1=pdfn(x);
y1=pdfn(y);
z=x1.*y1';

xyz(x,y,z);
surface(x',y,z);
contour(x',y,z);
```

Graphical images can be printed from within a program or saved to disk in one of the file formats supported using the graphtprt procedure. First, consider printing a GAUSS graph. Suppose the graph has already been drawn using one of the graphic procedures, for example xy. Then the graph is printed to a printer with the settings specified in pqgrun.cfg by the two commands

```
graphprt("-p");
rerun;
```

The rerun-command redraws the picture. The procedure graphprt has a string as its argument and certain flags are set in that string. These flags apply to the first graph drawn after the call to graphprt. The most important flags are -p (print the graph using the settings in pqgrun.cfg), -c=k (convert file format to (k=1) Encapsulated Postscript, (k=2) Lotus PIC-format, (k=3) Hewlett Packard Graphics Language HPGL format, or (k=4) coloured PCX bitmap), and -cf=test.ext (save the graphic file as test.ext). The following two commands would save the graphic last shown as Encapsulated Postscript file with name figure.eps in the subdirectory c:\tex\docs\graph:

```
graphprt("-c=1 -cf=c:\\tex\\docs\\graph\\figure.eps");
rerun;
```

Note the double backslashes in the filename. Files can also be converted from the window that shows the graphic. Using File/Convert/ from the menu bar one is presented a list with formats that the file can be converted to, see figure 8.2. The name of the file depends on the setting of the variable cvt_filename in the file pqgrun.cfg. If this variable is commented out (by putting a # in front of it), the name will be cvt_<nnn>.<ext>, where <ext> indicates the type of conversion (eps for Encapsulated Postscript, pic for Lotus .pic, hpg for HPGL, and pcx for PCX). Files are numbered sequentially, so the first converted file would be named as cvt_000.eps, the second would be cvt_001.eps, etc.

8.2 Incorporating GAUSS Graphics in Text Documents

8.2.1 GAUSS Graphics in Microsoft Word Documents

A utility PlayW to change graphic formats can be downloaded from the Aptech website. At the time of writing, though, this utility is not completely functional. Hence, it is not yet possible to cut-and-paste GAUSS graphics from a graphics window into a Windows wordprocessor. It is easily possible though to include GAUSS graphics in a Microsoft Word document. First, the graphic has to be saved in HPGL-format, either using the graphprt-command or from the File/Convert-menu. An HPGL-file can be read into Word if the appropriate filter has been installed. That filter (with installation instructions) can be downloaded free of charge, either from the Mi-

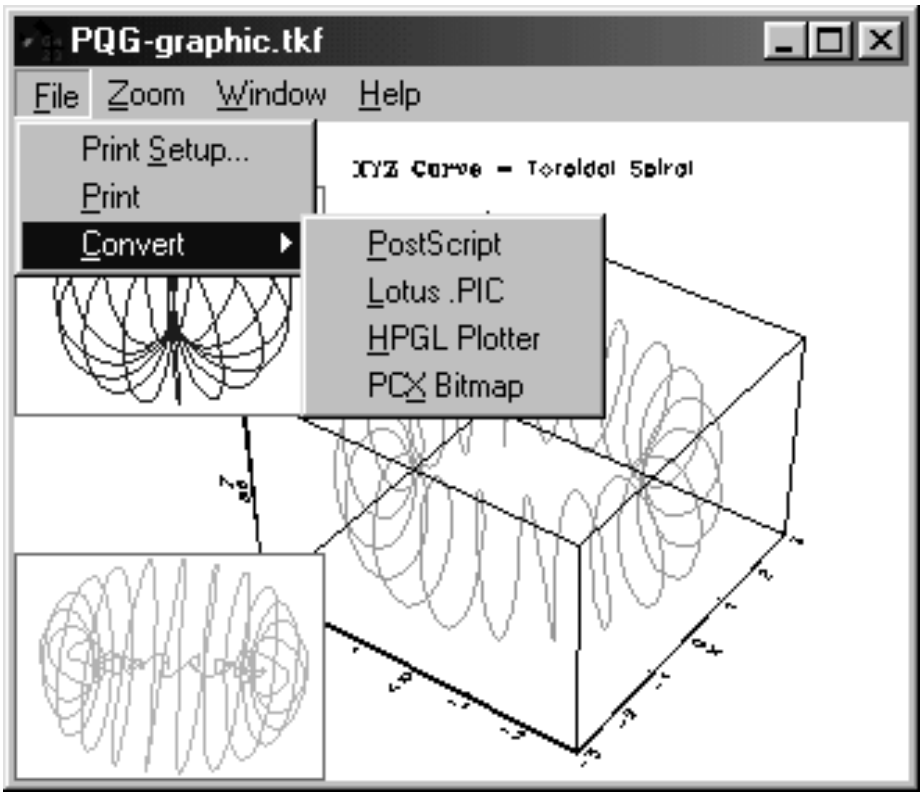


Figure 8.2: Graphic format conversion from the graphic window

crosoft website[1] or from the website of the author[2]. An HPGL-file can now be inserted using 'Insert/Picture/From file'. Note that the filter assumes extension `hgl` for HPGL-files, so it will not list files saved with extension `hpg` as GAUSS does by default. Hence, to choose an HPGL-file to be read in one should look for 'all files' and pick the appropriate HPGL-file. Since Word does not know which filter to use for files with extension `hpg`, the user is prompted for the format of the file. After choosing 'HP Graphics Language' the file is read into the Word document.

8.2.2 GAUSS Graphics in LaTeX2e Documents

It is not possible to cut-and-paste GAUSS graphics into LaTeX2e-documents. However, it is simple to include GAUSS graphics in a LaTeX2e-document. The easiest way to incorporate GAUSS graphics in LaTeX2e documents is to save the graph in Encapsulated Postscript format (`.eps`) and to read it in the LaTeX document with the `graphicx`-package[3]. A Postscript file can be read into a LaTeX2e-document as in

```
\includegraphics[width=\textwidth]{c:/tex/docs/graph/figure.eps}
```

The optional parameter `width` is used to set the width of the picture as it appears in the document. Another useful parameter is `angle` which is used to rotate the picture. An example of a graphics file read into a LaTeX2e-document is given in given in figure 8.3. Note that the background of the picture, which is black when the picture is shown on screen, has been changed to a white background.

[1] http://support.microsoft.com/support/kb/articles/Q196/5/06.asp

[2] `www.rhkoning.com/gauss`

[3] This booklet was typeset using MiKTeX, that is available free of charge at `ftp.dante.de` in the subdirectory `win32/miktex`. All graphics in this book were included as described in this section.

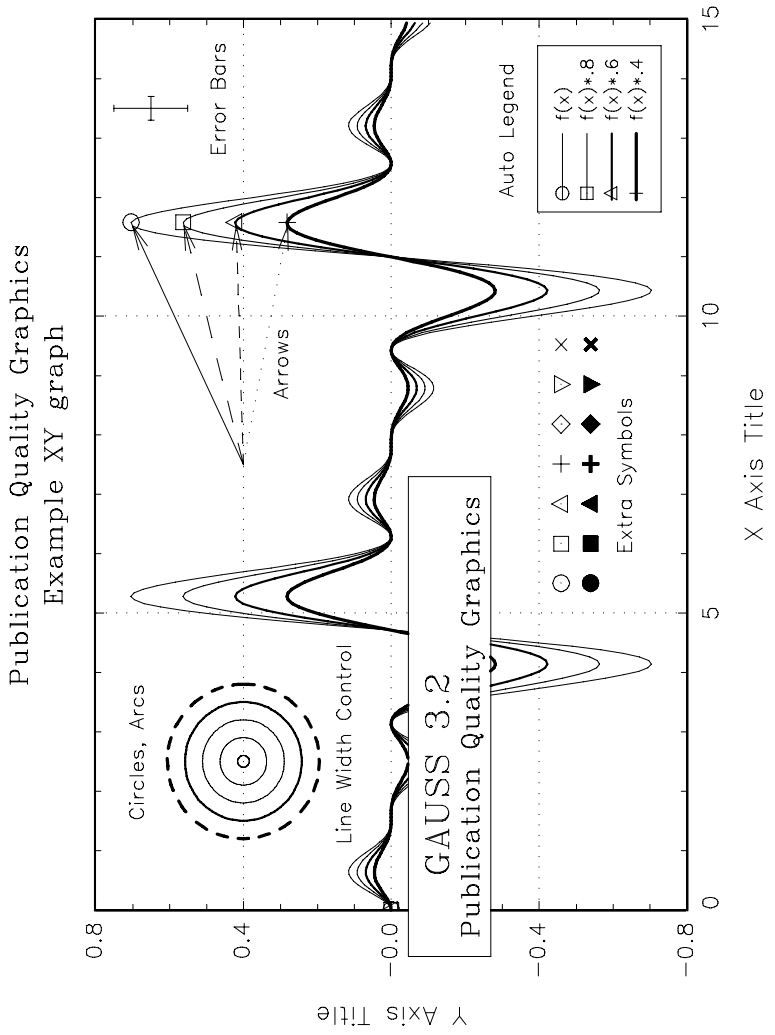


Figure 8.3: Graph from `examples/pxy.e`

Chapter 9

Two Examples

In this chapter we give two examples of programs written in GAUSS. In the first example a library that calculates kernel estimates is discussed. In the second example the finite sample properties of the ML-estimator in a probit model are derived by means of a simulation study[1].

9.1 A Kernel Estimation Library

Kernel estimation is a technique to estimate the density function of a random variable. Suppose we have n independently identically distributed observations $x_1, \ldots, x_n$ from an (unknown) density function $f(x)$. The density function may be estimated parametrically by assuming some functional form that depends on a few parameters which are estimated. Alternatively, it may be estimated non-parametrically, for instance by constructing a frequency distribution or a kernel estimate, which is basically a smoothed version of a frequency distribution. A standard reference on density estimation is Silverman (1986). We follow his notation in this chapter. The kernel library implements five procedures for kernel estimation of unknown densities. The first procedure (`ukernel`) estimates a univariate kernel according to

$$\hat{f}(x) = \frac{1}{nh} \sum_i K\left(\frac{x - z_i}{h}\right) \tag{9.1}$$

In this equation, x is the point where the kernel is to be evaluated, z_i ($i = 1, \ldots, n$) are the observations, h is the bandwidth, and $K(\cdot)$ is the kernel function. The second procedure estimates a multivariate kernel:

$$\hat{f}(x) = \frac{1}{nh^d} \sum_i K\left(\frac{1}{h}(x - z_i)\right) \tag{9.2}$$

[1] All GAUSS code of these examples can be downloaded from `www.rhkoning.com/gauss`.

k_epan	Epanechnikov kernel	$\frac{15}{16}(1-x^2)^2 I_{[-1,1]}(\mid x \mid)$
k_gauss	Gaussian kernel	$\frac{1}{\sqrt{2\pi}} \exp(-\frac{x^2}{2})$
k_triang	triangular kernel	$(1- \mid x \mid) I_{[0,1]}(\mid x \mid)$
k_rect	rectangular kernel	$\frac{1}{2} I_{[-1,1]}(\mid x \mid)$

Table 9.1: Kernel functions in kernel

Here, z and x are vectors and d is the dimenstion of the density to be estimated. The third procedure computes the Nadharaya-Watson kernel regression estimator:

$$\hat{m}(x) = \frac{\sum_i K\left(\frac{1}{h}(x - z_i)\right) y_i}{\sum_i K\left(\frac{1}{h}(x - z_i)\right)} \tag{9.3}$$

Finally, two auxiliary procedures are implemented, one is used to calculate the bandwidth (bandw1, $h = 0.9n^{-1/5}\min(\sigma, IQR/1.34)$ with IQR the interquartile range of the data and σ the standard deviation of the data, see Silverman (1986), p. 47) and one procedure (view_ukernel) to display the results of the univariate kernel estimation procedure. Besides these main procedures, four different kernel functions are implemented, see table 9.1.

```
/* kernel.src version 1.0
Source file with functions of the kernel library:
    ukernel     procedure for univariate kernel estimation
    mkernel     procedure for multivariate kernel estimation
    bandw1      procedure for automatic bandwidth selection
    nw          procedure for nonparametric regression
    viewuknl    procedere for plotting the estimated density

*/

/* comment the following line out if you don't have the pgraph
library installed. You will not be able to use the procedure
view_ukernel */

#include pgraph.ext

/* procedure to calculate a univariate kernel density estimate
and its
derivative
usage:      {f, d, h} = ukernel(x, z, h, w, &kf);
input:      x:      T vector where density is to be estimated
```

```
            z:      n vector with observed data points
            h:      scalar bandwidth, if h<=0 bandwidth is
                    determined by procedure bandw1(z)
            w:      n vector with weights
            &kf:    pointer to weighting function
output:     f:      T vector with estimated density
            d:      T vector with estimated derivative
            h:      scaler bandwidth
*/

proc (3)=ukernel(x, z,h,w, &kf);
 local arg, i, n, f, d, k, kff, kfd, pkff,argl,argh,kffl,kffh,
  fl,fh,hl,hh;
 local kf:proc;

 /* error checking here */
 if (cols(x)>1);
  errorlog "ukernel.g: x has too many columns";
  retp(-1,-1,-1 );
 endif;
 if (cols(z)>1);
  errorlog "ukernel.g: z has too many columns";
  retp( -1,-1,-1 );
 endif;

 /* initialization */
 i = 1;
 n = rows(x);
 k = cols(x);

 /* determine bandwidth */
 if (h<=0);
  h=bandw1(z);
 endif;

 f=zeros(n,1);
 d=zeros(n,1);
 do while (i<=n);
  arg = (x[i]-z)/h;
  {kff, kfd} = kf(arg);
  f[i] = meanc(kff.*w)/h;
  d[i] = meanc(kfd.*w)/(h^2);
  i=i+1;
 endo;
```

```
 retp( f,d,h );
endp;


/* procedure to calculate a multivariate kernel density estimate
 and its derivative
usage:      {f, d, h} = mkernel(x, z, h, w, &kf);
input:      x:      T x k matrix where density is to be estimated
            z:      n x k matrix with observed data points
            h:      scalar, bandwidth parameter (same bandwidth
                    for all components), if h<=0 it is determined
                    by eq 4.14 of Silverman
            w:      n vector with weights
            &kf:    pointer to weighting function
output:     f:      T vector with estimated density
            d:      T x k matrix with estimated derivatives
            h:      scalar, bandwidth used
*/

proc (3)=mkernel(x, z, h, w, &kf);
 local arg,i,n,f,d,k,kff,kfd,pkff,p,deneq0;
 local kf:proc;

 i = 1;
 n = rows(x);
 k = cols(x);
 if (h<=0);
   p=1/(k+4);
   h=(4/(k+2))^p*n^(-p);
 endif;
 f = zeros(n,1);
 d = zeros(n,k);
 do while (i<=n);
  arg = (x[i,.]-z)/h;
  {kff, kfd} = kf(arg);
  pkff = prodc(kff');
  f[i] = meanc(pkff.*w)/(h^k);
  deneq0=kff .==0;
  kff=kff+deneq0;
  d[i,.] = meanc((kfd.*pkff.*w)./kff)'/(h^(k+1));
  i = i+1;
 endo;
 retp( f,d,h );
endp;
```

```
/* computes Nadaraya-Watson kernel regression estimator

usage:  {g,d,h} = nw(x,z,y,h,&kf);
input:  x:          T-vector where regression function is
                    evaluated
        z:          n-vector with data on independent variable
        y:          n-vector with data on dependent variable
        h:          scalar, bandwidth, if h<=0 the bandwidth is
                    determined automatically using procedure
                    bandw1
        &kf:        pointer to univariate kernel density estimator
output: g:          T-vector with values of regression evaluated
                    in x
        d:          T-vector with estimated derivative in x
        h:          scalar bandwidth used
*/

proc (3)=nw(x, z, y,h,&kf);
    local num, numder, denom, denomder, g, s, n, k, d,v;
    local kf: proc;

 n = rows(x);
 k = cols(x);
 g = zeros(n,1);
 d = zeros(n,1);

 if (k ne 1);
  print "error in nw.g: too many columns in x";
  retp( g, d );
 endif;
 if (cols(z) ne 1);
  print "error in nw.g: too many columns in z";
  retp( g, d );
 endif;

 {num,numder,h} = ukernel(x,z,h,y,&kf);
 {denom,denomder,h} = ukernel(x,z,h,1,&kf);
 s = (denom.==0);
 g = num./(denom+s);
 d = numder./(denom+s) - num.*denomder./(denom^2+s);
 retp( (1-s).*g, (1-s).*d,h );
endp;
```

```
/* bandw1
procedure to calculate the optimal bandwidth in kernel estimation
of a density. The optimal bandwidth is calculated according to
eq. 3.31 of Silverman (1986)

usage:      h=bandw1(y);
input:      y:      n-vector whose density will be estimated;
output:     h:      scalar, optimal bandwidth choice;

*/

proc bandw1(y);
 local s, ys,n, a, iqr, qi1, qi3;

 if (cols(y)>1);
  errorlog "input error in bandw1.g: too many columns";
  retp( -1 );
 endif;
 s=sqrt(vcx(y));
 n=rows(y);
 ys=sortc(y,1);
 qi1=round(0.25*n);
 qi3=round(0.75*n);
 iqr=ys[qi3]-ys[qi1];
 retp( 0.9*minc(s|(iqr/1.34))/n^0.2 );
endp;

/* view_ukernel
procedure to plot kernel density estimate. Library pgraph must
be activated.

usage: call view_ukernel(x,f,h);
input:      x:  n-vector with data points
            f:  n x k matrix (k=1 or k=3) with estimated densities
                for different bandwidths
            h:  k-vector (k=1 or k=3) with bandwidths
output:     none
globals:    all globals of the pgraph library

*/

proc (0)=view_ukernel(x,f,h);
 local data,k,xlow,xhigh,flow,fhigh,xlegend,ylegend;
```

```
 /* error checking */
 if (cols(f) ne rows(h));
  errorlog "error in viewkrnl.g: rows(f) unequal to cols(h)";
  retp();
 endif;
 /* set global variables pgraph */
 _pdate=0;

 k=cols(f);
 xlow=minc(x);
 xhigh=maxc(x);
 flow=minc(minc(f));
 fhigh=maxc(maxc(f));
 xlegend=xlow;
 ylegend=flow+0.8*(fhigh-flow);
 _plegctl=1|4|xlegend|ylegend;
 if (k==1);
  _plegstr="h=" $+ ftos(h,"%*.*lf",5,3);
  else;
  _plegstr="h=" $+ ftos(h[1],"%*.*lf",5,3) $+ "\Oh="
   $+ftos(h[2],"%*.*lf",5,3)$+"\Oh="$+ ftos(h[3],"%*.*lf",5,3);
 endif;
 data=sortc(x~f,1);
 xy(data[.,1],data[.,2:cols(data)]);
endp;




/*
This file contains some kernel functions. All functions take an
n x k matrix u as their argument and return an n x k matrix with
the function evaluated in each point of x and an n x k matrix d
with the derivative of the function in each point of x.
    k_bw:     biweight kernel function
    k_epan:   Epanechnikov kernel
    k_gauss:  Gaussian kernel
    k_triang: triangular kernel
    k_rect:   rectangular kernel

usage:
    {f,d}=k_bw(u);
*/
```

```
proc (2)=k_bw(u);
 local select;
 select = abs(u).<=1;
 retp( 15/16*((1-u.^2).^2).*select, -15/4*u.*(1-u.^2).*select);
endp;

proc (2)=k_gauss(u);
    retp( pdfn(u), -u.*pdfn(u) );
endp;

proc (2)=k_epan(u);
    local c, s;
    s = abs(u).<sqrt(5);
    c = 0.75/sqrt(5);
    retp( c*(1-0.2*u.^2).*s, -0.4*c*u.*s );
endp;

proc (2)=k_rect(u);
    retp( 0.5*(abs(u).<1), 0*u );
endp;

proc (2)=k_trian(u);
    local s, a;
    a = abs(u);
    s = a.<1;
    retp( (1-a).*s, (-(u.>=0) + (u.<=0)).*s );
endp;
```

The following example is a program that illustrates some of the procedures of the library,. In this example, a standard normal density is estimated using different kernel functions.

```
/* in this example we draw 1000 random numbers from a normal
distribution and we estimate the density using the kernel
library. Three different graphs will be created using different
kernel funtions. Note that the pgraph library must be activated
in order to use view_ukernel.
*/
new;
library kernel,pgraph;
```

```
nobs=1000;

r=rndn(nobs,1);

x_axis=seqa(-3,6/1000,1000);
{f1,d1,h1}=ukernel(x_axis,r,0,1,&k_gauss);
call view_ukernel(x_axis,f1,h1);
{f2,d2,h2}=ukernel(x_axis,r,0,1,&k_epan);
call view_ukernel(x_axis,f2,h2);
{f3,d3,h3}=ukernel(x_axis,r,0,1,&k_rect);
call view_ukernel(x_axis,f3,h3);
end;
```

9.2 Finite Sample Properties of the ML-estimator in Tobit Model

In the Tobit-model, data are generated according to

$$y_i^* = \beta' x_i + \epsilon_i, \qquad \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$
$$y_i = \max(0, y_i).$$

Only y_i and x_i are observed. If the fraction of observations with $y_i = 0$ is very small, the model resembles the standard linear model with normal disturbances. However, if the fraction of zeros is non-negligible, the model cannot be estimated using ordinary least squares. Consistent estimates of the parameters can be obtained using maximum likelihood. The finite sample properties of this estimator is unknown in this model, asymptotically the estimator has a normal distribution. In order to assess whether the asymptotic distribution provides a reasonable approximation to the finite sample distribution, one can perform a small simulation study. In this example, 100 samples of 50 observations are drawn, and for each sample the parameters of the model are estimated. Two sets of endogenous variables are generated, in model 1 approximately 16% of all observations on the endogenous variable are 0, in model 2 this percentage is 2.2%. We would expect the estimates of model 2 to conform more closely to the standard normal distribution than those of model 1.

```
library maxlik,kernel,pgraph;
maxset;

output file=tobitsim.out reset;

nobs=50;
nrep=100;
```

```
beta1=1|-1|0.5|-0.25;
beta2=3|-1|0.5|-0.25;
x=ones(nobs,1)~3*rndn(nobs,3);



_max_GradProc=&loglikgd;
_max_GradCheckTol=1e-3;


let b1={};
let b2={};
i=1;
do while (i<=nrep);
 y1=x*beta1+2*rndn(nobs,1);
 y2=y1+2; /* same disturbances as in y1 */
 y1=(y1.>0).*y1;
 y2=(y2.>0).*y2;
 {x0,f,g,cov,retcode}=maxlik(x~y1,0,&loglik,beta1|2);
 if (retcode==0);
  b1=b1|x0[1:4]';
 endif;
 {x0,f,g,cov,retcode}=maxlik(x~y2,0,&loglik,beta2|2);
 if (retcode==0);
  b2=b2|x0[1:4]';
 endif;
 i=i+1;
endo;

print "fraction succesful convergences model 1";
rows(b1)/nrep;
print "mean b1/max b1/min b1/var b1/beta1";
meanc(b1)~maxc(b1)~minc(b1)~diag(vcx(b1))~beta1;
print "fraction succesfull convergences model 2";
rows(b2)/nrep;
print "mean b2/max b2/min b2/var b2/beta2";
meanc(b2)~maxc(b2)~minc(b2)~diag(vcx(b2))~beta2;

grid=seqa(-3,6/200,200);
sb12=sqrt(vcx(b1[.,2]));
sb22=sqrt(vcx(b2[.,2]));
{f1,d1,h1}=ukernel(grid,(b1[.,2]-beta1[2])/sb12,0,1,&k_epan);
{f2,d2,h2}=ukernel(grid,(b2[.,2]-beta2[2])/sb22,0,1,&k_epan);
_plegctl={ 2 4 1 5 };
```

```
_plegstr="model 1\000model 2\000standard normal";
call xy(grid,f1~f2~pdfn(grid));
graphprt("-c=1 -cf=e:\\intro99\\pictures\\tobitsim.eps");
rerun;

end;

proc loglik(theta,z);
 local y,x,b,s;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 retp( (y.==0).*ln(cdfn(-x*b/s)) +
  (y./=0).*(-0.5*ln(s^2)-0.5*(y-x*b)^2/s^2) );
endp;

proc loglikgd(theta,z);
 local y,x,b,s,e,gb1,gs1,gb0,gs0,a;
 x=z[.,1:cols(z)-1];
 y=z[.,cols(z)];
 b=theta[1:cols(x)];
 s=theta[cols(x)+1];
 e=y-x*b;
 a=-x*b/s;
 gb0=-(pdfn(a)./cdfn(a)).*x/s;
 gs0=-(pdfn(a)./cdfn(a)).*a/s;
 gb1=e.*x/s^2;
 gs1=-1/s+e^2/s^3;
 retp( ((y.==0).*gb0+(y./=0).*gb1)~((y.==0).*gs0+(y./=0).*gs1) );
endp;
```

The result of this simulation is shown in figure 9.1. We do not wish to discuss Monte Carlo simulation in detail, but to demonstrate that it can be straightforward to perform a small simulation study in GAUSS.

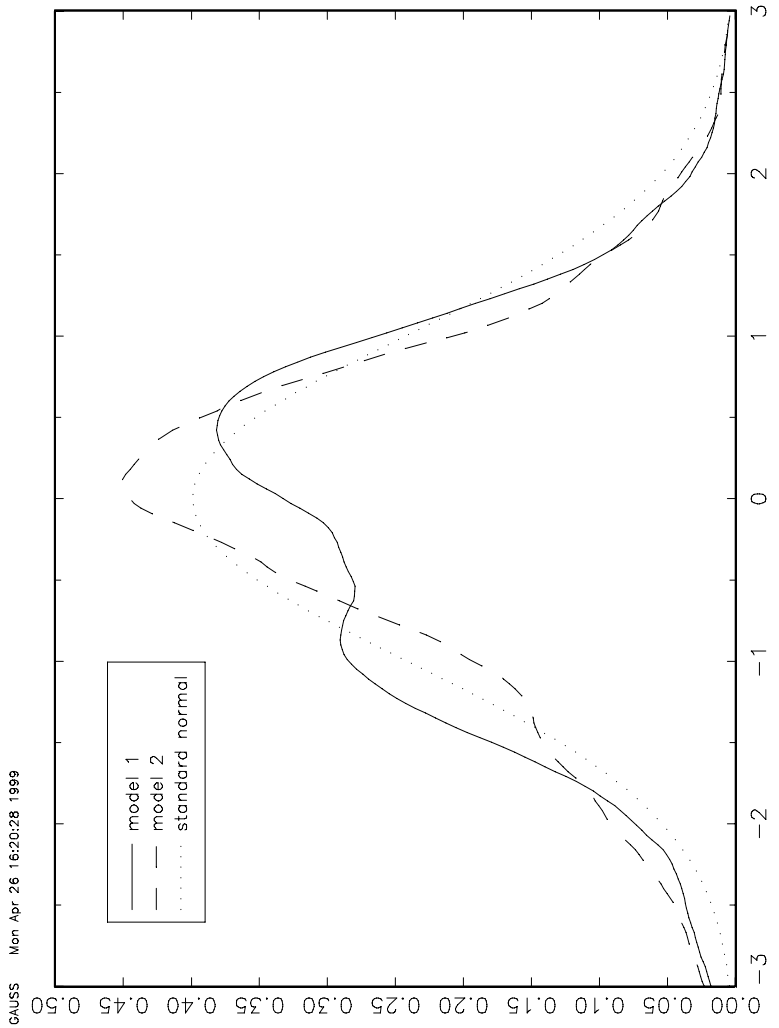


Figure 9.1: Distribution of ML-estimator

Chapter 10

Exercises

1. Check your setup of GAUSS by running some of the example files in the `gauss\examples` directory. Have problems fixed by your system administrator.
2. Make a subdirectory for the programs and procedures you'll write during this course. Ensure that it is in the search path of GAUSS. Modify the file `startup` such that the working directory is set to this directory as soon as GAUSS is started.
3. Write a program that initializes a matrix with ones, a matrix U with uniform random numbers and one matrix with keyboard input. Count and print the number of elements of U that are smaller than 0.5.
4. Write a program that converts your day of birth (a string variable like `birth = "July-04-1956";` to a numerical vector with 3 elements, the first element representing the month, the second representing the day and the third representing the year.
5. Create a 1000-vector with $\mathcal{N}(1,4)$-distributed random variables and calculate the fraction of numbers smaller than 0, between 0 and 2, and exceeding 2. Use GAUSS to calculate the probabilities of these events and compare them with the fractions found.
6. Write a procedure that determines the Euclidean length of a vector x.
7. Write a procedure that calculates the OLS estimator in the linear regression model, its covariance matrix, and an estimate for the residual variance.
8. Write a procedure that has two arguments (a matrix and a string with a file name) and that save the matrix to an ASCII file with the specified file name.

9. In panel data econometrics one usually stacks the observations such that the individual index runs slow and the time index runs fast:

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_T \end{pmatrix}.$$

Here, y_1 is an N-vector with the observations for all individuals in period 1, y_2 is an N-vector with all observations in the second period, etc. Write a GAUSS program that re-orders the vector y such that the time index runs fast and the individual index runs slow, *i.e.*

$$\tilde{y} = \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \vdots \\ \tilde{y}_N \end{pmatrix}.$$

Here, $\tilde{y}_1$ is a T-vector with all observations for individual 1, etc.

10. Write a procedure that calculates an N-vector with contributions to the loglikelihood in a Tobit model with N observations. Minimize the amount of computations required.

11. Write a program that generates data according to the probit model:

$$\begin{aligned} y_i^* &= \beta' x_i + \varepsilon_i \qquad \varepsilon_i \sim \mathcal{N}(0,1) \\ y_i &= \begin{cases} 0 & y_i^* \leq 0 \\ 1 & y_i^* > 0 \end{cases} \end{aligned} \tag{10.1}$$

Only the x_i's and y_i are observed. Save the observations in a dataset.

12. Using this dataset, calculate descriptive statistics and estimate β using OLS.

13. Create a graph of the standard normal, t- and Cauchy-density functions. Save your graph and generate a print.

14. Write a procedure that calculates the contributions to the loglikelihood of a probit model. The first parameter of the procedure is the parameter β and the second part is a datamatrix Z.

15. Examine by means of a simple simulation study whether Glesjer's test for the presence of heteroscedasticity in a linear regression model is more powerful than an omnibus test like the Breusch-Pagan test (see Greene (1993), p. 394–396).

BIBLIOGRAPHY

Dennis, J.E. and R.B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs: Prentice Hall.

Greene, W.H. (1993). *Econometric Analysis* (third ed.). New York: MacMillan.

Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery (1992). *Numerical Recipes in C* (Second ed.). Cambridge: Cambridge University Press.

Silverman, B.W. (1986). *Density Estimation*. London: Chapman & Hall.

White, H. (1982). Maximum likelihood estimation of misspecified models. *Econometrica 50*, 1–25.

APPENDIX A

CONVERTING ASCII TO GAUSS DATASETS

An ASCII dataset has to be converted to a GAUSS dataset before it can be used. To facilitate this conversion, the utility `atog.exe` has been provided. This utility can be operated both interactively and in batch mode. In case one uses the utility interactively, it is started by `atog` at the DOS-command prompt. The utility returns with a prompt \ after which a limited number of commands can be entered (see below). All commands are terminated with a semicolon (`;`). An interactive session is ended by the command `quit`.

In batch mode, the conversion to a GAUSS dataset is performed by `atog convert.cmd`. The file `convert.cmd` contains the commands that perform the conversion. The most useful commands[1] are

append append ASCII dataset to an existing GAUSS dataset,

help display help (interactive mode only),

input parameters name of the ASCII dataset,

output parameters name of the GAUSS dataset,

msym parameter missing value character (default dot '`.`'),

invar parameters specification of data in ASCII dataset,

outvar parameters list of names of variable to be saved in GAUSS dataset,

run execute commands entered (interactive mode only).

The most crucial command is the `invar`-command. This commands tells `atog` how the data are organized in the ASCII dataset. ASCII datasets may either be softly delimited (elements are separated from each other by tabs,

[1] For a detailed description we refer to the GAUSS manual.

spaces, or carriage/returns-line feeds), hard delimited (elements are delimited by a printable character) or not be delimited at all (packed ASCII). In the first case, the command reads

```
invar $name #age weight $char[1:10] #numvar[03];
```

The first variable is a character variable (indicated by the $-sign), the second and third variable are numerical variables (indicated by the #-sign), the next ten variables are labeled char1 to char10, and finally, the last three variables are numerical again and labeled as numvar01, numvar02 and numvar03.

If the ASCII dataset is hard delimited, the format changes to something like

```
input data.csv;
output datacsv;
invar delimit(,,N) #var1,var2,var3,var4,var5;
outvar var1,var2,var3,var4,var5;
```

This command file converts the dataset

```
1,2,3,4,5,
6,7,8,9,10,
```

The keyword delimit has two optional parameters: the first is the delimiter (, in this example. One uses \r if each line contains one record and the variables are separated by commas) and the second one indicates whether the last element of an observation is followed by a delimiter (in which case it takes the value N). atog386 assumes that the last variable is not followed by a delimiter if no second argument is specified.

Note that the values taken character variables may not include spaces. atog386 reports an error if it encounters an input line like

```
34,s korea,belgium,0,0,32
```

because of the space in the second field. If the data are organized as packed data with fixed records length there are no problems if a character variable takes a value with a space in it.

A packed ASCII file must have records with fixed lengths. The keyword record is used to indicate the length of the record (including the final CR/LF, which takes two positions). Furthermore, the format of the variables must be specified:

```
invar record=47 $(1,4)name #(5,2)age (7,4.2)weight
 $(*,1)char[1:10]  #(*,8.2)numvar[03];
```

The first variable (name, a character variable) starts at position 1 and has length 4. The age variable (numerical) starts at position 5 and has length 2. The starting position of the weight variable is 7, it occupies 4 bytes, of which the last two are decimals. A decimal point is inserted between the second and third element. The next ten variables are all character variables of length 1. The asterisk in the format field indicates that the first field of that variable is the one immediately after the last field of the previous variable, so that no position in the data file are skipped. The last three variables are numerical, have length 8 and the last two numbers are converted to decimals. A record has 45 fields and is terminated by a <CR><LF>, so the total length is 47 bytes.

An example of a complete command file for atog386 is

```
input c:\gauss\data\wbo.asc;
output c:\gauss\data\wbo;
invar huur inkomen ihs respnr;
outvar huur inkomen ihs respnr;
msym &;             @ missing symbol@
```

Note that no double backslashes are used in naming both the input and the output file, the variables invar and outvar are literals. Comments are enclosed in @-signs. The command atog386 convert.cmd converts the ASCII file wbo.asc to the GAUSS dataset wbo.dat and accompanying header file wbo.dht.

APPENDIX B

GAUSS LIBRARIES

GAUSS is available on the following platforms: MSDOS, Windows95/98/NT, DEC UNIX, HP-UX, IBM RISC, Linux, OS/2, SGI IRIX and Solaris. The graphical user interface differs between these implementations, the language itself does not differ (except for the graphics commands).

The following GAUSS-libraries are distributed by Aptech Systems:

- Constrained Maximum Likelihood
- Constrained Optimization
- Curvefit
- Descriptive Statistics
- FANPAC
- Linear Programming
- Linear Regression
- Loglinear Analysis
- Maximum Likelihood
- Nonlinear Equations
- Optimization
- Quantal Response
- Time Series

Except for FANPAC, which runs under UNIX and Windows operating systems only, all these libraries are supported on all platforms.

Apart from these modules some third-party modules are available as well:

- CellVision (DOS)
- COINT (DOS, OS/2, UNIX, Windows)
- CTRLGAUSS (DOS, Windows)
- DATAWIZ (DOS, OS/2, Windows)
- FAIR-TAYLOR (DOS)
- Fastsort (DOS, UNIX)
- GAUSSX (DOS, UNIX, Windows)
- LALIB-386 (DOSm Windows)
- LINCS (DOS, OS/2, UNIX, Windows)
- MARKOV (DOS, Windows)
- Mercury (Windows)
- Mercury GE (Windows)
- MISS (DOS, OS/2, UNIX, Windows)
- Quadratic Programming (DOS, UNIX, Windows)
- QUEGAUSS (DOS, Windows)
- SimGAUSS (DOS, Windows)
- SNAP (DOS)
- SSATS (DOS, OS/2, UNIX, Windows)
- Stat/Transfer UNIX, Windows)
- TSAGL (DOS)
- TSM (DOS, OS/2, Windows)

More information on these modules can be found at the WWW-sites of Aptech Systems (`www.aptech.com`) and on those of other GAUSS distributors.

APPENDIX C

PROGRAM CODE

Computer code for the programs in this book can be downloaded from `www.rhkoning.com/gauss`. The archive file that contains the code is called `bookcode.zip`. The files in that archive correspond to the programs in the book according to the table below.

page 18	`chap3\probit1.prg`
page 22	`chap4\vectoriz.prg`
page 23	`chap4\if-exam.prg`
page 25	`chap4\boxcox1.g`
page 26	`chap4\boxcox2.g`
page 27	`chap4\danger.prg`
page 28	`chap4\numder.prg`
page 30	`chap4\ols.prg`
page 34	`chap5\course1.lcg`
page 35	`chap5\testlib3.src`
page 36	`chap5\testlib3.dec`
page 37	`chap6\file.asc`
page 40	`chap6\datasets.prg`
page 41	`chap6\readdata.prg`
page 44	`chap7\maxlik1.prg`
page 48	`chap7\maxlik2.prg`
page 50	`chap7\maxlik3.prg`
page 54	`chap8\density.prg`
page 56	`chap8\hist.prg`
page 57	`chap8\3d.prg`
page 64	`chap9\kernel.srcg`
page 69	`chap9\kfunctio.srcg`
page 70	`chap9\kernel1.e`
page 71	`chap9\tobitsim.prg`

Table C.1: Files in the code archive