Gauss Tutorial

Pierre Giot and Jean-Pierre Urbain
Department of Quantitative Economics
Maastricht University

First draft: November 1994
This version: November 2000

Contents

1 Introduction

This manuscript contains a brief introduction to the use of the software package GAUSS.[1] GAUSS is a statistical package used for statistical and econometric purposes and is best suited for data analysis, estimation, testing, forecasting, simulation as well as professional quality data graphing. These notes are intended to be supplementary to the official GAUSS manuals and to show some of the basic principles of programming using a matrix language rather than being an exhaustive presentation of the possibilities offered by GAUSS. Only some of the most fundamental parts of GAUSS are explained herein. For a detailed presentation of the numerous possibilities offered by GAUSS we advise the student to read the corresponding section of the GAUSS manuals. Volume 1 of the manual (*System and Graphics* manual) treats all topics related to the GAUSS system and the Graphics Interface. Volume 2 of the manual (*Command Reference* manual) provides a useful indexed command reference guide. It is important to understand that the current notes are not sufficient to work efficiently in GAUSS nor are they aimed to provide an alternative to a careful use of the manual and the reference guide on which these notes are entirely based. Also, when using particular application modules (see below), it is important to study carefully the corresponding reference manual.

GAUSS is a programming language designed to operate with and on matrices. In comparison to other popular packages used in statistics and econometrics such as RATS, MICROTSP, SAS, SPSS, PCGIVE, MICROFIT, etc., GAUSS is characterized by its extreme flexibility enabling the user to handle both non-standard and standard problems in statistics and econometrics. Current competitors to GAUSS are essentially MATLAB and OX. The major advantage of GAUSS, and his competitors, is that it allows you to do almost everything you want with data. It has moreover almost become one of the standard in econometric research so that numerous routines (procedures) become easily available. Since it operates directly on matrices, it makes GAUSS more useful for economists than standard programming languages where the basic data units are all scalars. GAUSS programs and functions are all available to the user, and so the user is able to change them contrary to most menu driven pre-programmed packages. Most statistical/econometric problems can be handled in a convenient and efficient way. You can specify your own model, new estimators, test statistics, estimate complicated likelihood function, etc.

The price to pay for this flexibility is naturally that you have often to do

[1]GAUSSis a trademark of *Aptech Systems Inc.*, Maple Valley, USA.

the job by your own (write your own programs, procedures, estimators,...). There is often unlikely to be a simple procedure to do a simple econometric task readily at hand. Notice also that given GAUSS' high flexibility, one will also rapidly observe that GAUSS may be found to be too tolerant of sloppy programming which means it is difficult for the computer to tell when mistakes occur. Nevertheless, numerous compiled procedures and intrinsic functions are already available. Also, some more advanced commands, procedures and functions are given in various applications modules. GAUSS is both a compiled language and an interpreter. It is a compiled language because it scans the entire program, translates it into (pseudo) binary codes and then executes the program. Since this (pseudo) binary code is not native of the CPU, GAUSS must interpret each instruction for the computer. GAUSS also allows you to compile programs that are very frequently used to a file that can be run over and over with no compile time.

For Internet users, it may be interesting to know that there exists a GAUSS software archive located at the American University in Washington D.C.. The site also contains links to various other sites of interest for GAUSS users. The Web address is:

`http://gurukul.ucc.american.edu/econ/gaussres/GAUSSIDX.HTM`

Other useful www addresses (these have been used as references for these notes):

- Online interactive GAUSS tutorial (nice!):

  `http://eclab.econ.pdx.edu/gpe/toc.htm`

- Excellent beginner's guide to GAUSS in Microsoft Word format (relates to an older version of GAUSS but much of the stuff is still valid):

  `http://scottie.stir.ac.uk/~fri01/gauss/gauss.html#seminar`

- MARK WATSON' on-line Gauss tutorial:

  `http://www.wws.princeton.edu:80/~mwatson/ec518/gauss_tutorial.html`

- GAUSS homepage:

  `http://www.aptech.com/`

2 Basic concepts

Let us first introduce some concepts that have to be clearly understood:

- **Expression.** An expression in GAUSS can be a matrix, string, constant, function or a reference to a procedure (or any combination) that returns a results with the assignment operator `'='`.
- **Statements.** A statement is a complete expression or command which will always end with a semicolon `';'` (except if we are in *command mode* in which case the semicolon can be omitted most of the time). If there is no assignment operator ('=') the expression is then an implicit `PRINT` statement. For example:

  ```
  y = x * 3 ;
  ```

  is a statement. The following two statements are equivalent:

  ```
  print y
  y;
  ```

  Remark: a statement is not always executable (example: the declaration of a matrix).
- **Programs.** A program is a structured set of statements that are run together.
- **Procedures.** A procedure allows you to define a new function that you create and which can then be used as if it was an *intrinsic* function. The procedures are isolated from the rest of the program (see below).
- **Library.** GAUSS allows you to create libraries of frequently used functions that the system will automatically find and compile whenever it finds a reference to it in a given program. When GAUSS encounters a symbol (or a procedure, or whatever,...) that has not previously been defined the `AUTOLOADER` of the system will automatically try to locate and then compile the files containing the symbol definition.[2] A GAUSS library serves as a *dictionary* to the source files that contain the definition of the symbol or of the procedure.

[2]The search path used by GAUSS is first the current directory, than those listed in the SRC_PATH variable in the Gauss.cfg file, that is the GAUSS configuration file.

- **Edit/Command modes.** The communication between the user and GAUSS is most efficiently handled through the editor and the command windows. To check whether the editor is active or not, just see the upper left corner of the menu bar that should display GAUSS-`edit`. The other mode is the `command` mode which is recognized by the presence of `(gauss)` at the beginning of a line. The *command mode (window)* is best used to create very short interactive (screen resident) programs, or simply for the execution of simple command lines such as displaying results of simple operations, displaying a matrix,... If you are within the Editor window and want to run the program you are working in and then jump to the command mode you just have to type `F3`. Conversely, from the command mode window you may switch to the edit window by typing `F4` or by using the menu bar which enables you to switch from one to the other.

- **Data types in** GAUSS. The two basic data types supported in GAUSS are **matrices** and **strings**. The latter can be used to store names of files, of variables, to specify messages to be printed,... Note that a matrix can have numeric or character elements. It is in fact not necessary to declare the type of a variable since this can change within a program (although it is better to respect the types of variables if possible). Matrices obviously include vectors (row and column) and scalars as sub-types, but these are all treated the same by GAUSS. For example:

  ```
  a = b + c;
  ```

  is valid whether `a,` `b`, and `c` are scalars, vectors, or matrices, assuming the variables are conformable. Note that the results of the operation might be different depending on the variable type. As noted above, matrices may contain numerical data or character data or both. Numerical data are stored in scientific notation to around 12 places of precision. Character data are sequences of *up to eight characters* which count as one element of the matrix. If you enter text of more than eight characters into the cells in a matrix, the text will be truncated.

  Strings are pieces of text of *unlimited* length. These are used to give information to the user. If you try to assign a string value to an element of the matrix, all but the first eight characters will be lost.

  Examples of data types

– Numerical 3 × 3 matrix

$$\begin{pmatrix} 1 & 2.2 & -3 \\ 6.2 & 9 * 10^{-6} & 5 \\ 7 & 9 & 99 \end{pmatrix}$$

– Character 2× 2 matrix

$$\begin{pmatrix} Will & Will \\ Harry & Steve \end{pmatrix}$$

– Strings

```
"Maastricht!"

"Strings may be pieces of text of long length"

"2.2"

" "
```

Remark the truncation of text in the character and mixed matrices. The null string " " is a valid piece of text for both strings and matrices. Since all matrix data are treated the same way, the user has sometimes to specify that GAUSS is dealing with character data. The `$` sign identifies text and is used in a number of places. For example, to display the value of the variable `var1` requires

```
print var1;
```

or

```
print $var1;
```

depending on whether `var1` is a numerical matrix, a character matrix or a string.

Note that all variables must be created and given an initial value before they are referenced to.

- **Notation, syntax and case sensitivity.** GAUSS could be described as a free-form structured language: structured because it is designed to be broken down into easily-read parts; free-form because there is no particular layout for programs. Notice that extra spaces between words are ignored. Commands are separated by a semi-colon. Program layout is generally a matter of personal choice and the user has freedom to lay out code in a style he finds acceptable. Acceptable names for variables are up to eight characters long. These may contain alphanumeric data and the underscore _ but should never start with a number. For example, acceptable names include

```
name Nam Nam1 nam_1 _name1 n_a_m_e
```

but

```
1eric 100 12_names if
```

are not accepted since the three first start by a number and the fourth name is actually a reserved name. Remark that GAUSS does not distinguish between uppercase and lowercase, except inside double quotes "...".

- Comments can be easily introduced in a program: between `@...@` or `/* ...*/`. Contrary to the former, the latter kind of comments can be nested. Example:

```
@ This is a comment @
/* This is another comment that can be nested */
```

- **Starting** GAUSS. To start GAUSS under Win98 you simply have to double-click on the GAUSS short cut icon and you are automatically set in *command mode.*

- **Help.** Under Win98, the user has access to the on-line help by simply typing `F1` which is very useful as it contains most of what can be found in the command reference guide.

3 Editor/Command windows

Since the communication between the user and GAUSS will most of the time be handled through the full-screen editor window or the command window, we will briefly discuss the basic features of these. First, it is important to note that when you click on the GAUSS shortcut you will bring up the Command window.

3.1 The Command Window

The Command window is the window that first appears when you start GAUSS. Its main feature is a large scrollable text edit pane. This is the place where the GAUSS prompt appears for you to enter and run interactive commands. It is also where all GAUSS program input output takes place. When you start GAUSS you will automatically be in the *command mode/window* which is characterized by the presence of the following start character at the beginning of the line:

```
(gauss)
```

A program in *command mode* is then defined as the code from the end of the line on which the cursor is located with the start character `(gauss)`. Pressing the `Enter` key will make a program begin executing. It thus allows you to work *interactively* in GAUSS. As such, you can only execute one line of commands but you may also run existing programs,... Menus of interest within the command window are: **File, Edit, Search, Font, Options, Window, and Help.**

- **File | Edit...**: This part of the menu opens a dialog for selecting a file to edit. The dialog lets you navigate drives and directories, specify a filename pattern, and see the files in a directory. The file selected is added to the top of the Edit list and loaded into the Edit window, which is brought to the foreground.

- **File | Run...**: Opens a dialog for selecting a file to run. The file selected is added to the top of the Run list and the file is executed, bringing the Command window to the foreground.

- **File | Stop...**: Stops the currently executing GAUSS program. This can be very valuable if you've written a never-ending do-loop and have doubts on the correctness of your codes.

- **File | Exit Gauss...**: Exits GAUSS without verification.

- **Edit | Undo Cut Copy Paste**: Standard windows commands.

- **Search | Goto Find Replace Find Again Replace Again**: Standard windows commands.

- **Font | Select...**: Opens a font selection dialog which allows you to change the font in the text edit panel of the Command and Edit windows. The font setting is saved between GAUSS sessions.

- **Options | Edit...**: Includes controls to: Set the tab length. Set insert/overstrike mode. Turn text wrapping on/off.
- **Options | Program...**: Includes controls to: Turn default libraries on/off. Set autoload/autodelete state. Turn declare warnings on/off. Turn dataloop translation on/off. Turn translator line tracking on/off. Turn normal linetracking on/off. Set Compile to File state. These are more advanced options that will not be discussed in details during the course. You are advised to leave the default settings.

The windows also simply lets you move between the Command and Edit windows. The Command window will accumulate output unless you clear the screen. There is however *no* hot key for doing so. To clean the screen you have to use `CLS` command near the beginning of the program. Within the Command window you have to type `cls` after the GAUSS prompt and hitting enter. Notice that a few other keys are useful such as `F4` to switch to the GAUSS-Edit windows and `F3` to execute (run) a program.

3.2 The Edit Mode/Windows

The GAUSS *edit window* is by far the one that is the mostly used in GAUSS. It allows you to create small data files, edit existing files, write down your programs, execute them and modify what appears to be eventually necessary, rerun, modify, without having to quit GAUSS. To get in *edit mode* when you are in the default GAUSS *command window*, just use the menu bar which is useful to load existing program files for example or type `F4` to switch to the GAUSS-Edit windows. The Edit window is a text editor. The Edit window and the Command window have many controls in common. The most obvious difference is that the Command window has `Stop` (running program) controls and the Edit window has `Save` (file) controls. Most of the possibilities offered by the menu under the Edit mode are self-contained and standard.
File | Save as... Opens a dialog for saving the current contents of the Edit window under a new filename. The text being edited is saved to the new file, and the new filename is added to the top of the Edit list. You will find this useful for copying a program under a new name so you can use it as a template for creating a new program. If you want to create a new file that you will call `myfile`, you may more simply within the command window simply type

```
Edit myfile <Enter>
```

and then the GAUSS-edit window will automatically appear. You are then in *edit mode* and already in `myfile`. You can enter your data set or create new program as in any usual text editor. If you have already a saved copy of your program, let say `myfile.prg`, on your disk, just type

```
Edit myfile.prg <Enter>
```

and you are then in *edit mode* where you can modify your `myfile.prg` file. More simply, you may use the menu bar. Notice that this implies that your file is in the working directory of GAUSS. If you have your file located on a floppy (which is the A: drive), you have to introduce a path name:

```
Edit A:\myfile.prg <Enter>
```

which specifies where to go to get `myfile.prg.` Edit thus allows for path to get existing file edited. The build-in editor in GAUSS has a multitude of more or less standard Win98 editing keys or keystrokes that are useful when building or modifying a program.

3.3 Graphic Windows

When you have run a program in which creates graphs (whatever the type of graph), GAUSS will open a so-called PQG window, where PQG stands for Publication Quality Graphics. We will discuss the creation of graphics, this window and its options below briefly. When GAUSS creates a graph, it stores it in a file with a `.tkf` extension. The `tkf` files will be stored in your general GAUSS working directory. After you have created a graph you can save it and protect it from overlay by moving it to a folder or separate directory.

4 Loading data and creating data sets

There are basically two ways to get data into GAUSS: you may enter vectors and matrices directly into your program, or you can read data in from an existing file. Both methods have their uses. In this section we first concentrate our attention to the simple questions:

- How can you create a small data set in GAUSS?
- How can you read external data files in GAUSS?

Assume you are already in the GAUSS *command mode* and you would like to create your own data set consisting of 3 variables for which you have 5 observations on each of these. The easiest way to proceed is to type

```
Edit mydata <Enter>
```

Once you are under the GAUSS-Edit window, after the message `Editing mydata` has appeared on lower left corner of your the screen you may type (for example)

```
10.01  22.225  2.30
11.08  55.233  3.02
12.59  65.158  4.02
13.26  61.258  5.03
19.02  64.267  6.12
```

Remark: the different columns are the different variables and the different rows are the observations on these variables. Use then the menu bar to save these data in an ASCII file that is called `mydata`. Alternatively you may just type `F2`. Naturally such an ASCII file can also be created using an external editor or as the output of various software (Lotus, Excel, RATS,...).

The question is now how can we load this small data set so that we will be able to use it in GAUSS. Remember that GAUSS work with matrices so that the data file has to be loaded in a matrix. Let us assume that we call this matrix `x`. We can simply type the following lines:

```
load x[5,3] = mydata;
@ loads the data mydata into the matrix x @
print x; @ prints the matrix x @
```

`LOAD` accepts path names if your file `mydata` is not in the GAUSS directory. Various forms of `LOAD` are available, they do depend on the specific natures of the file you want to load (see *Command Reference Guide* or the on-line help for more information). The command `LOAD` can be used to read in data from an ASCII file (`*.asc, *.txt, *.prn,` or`*.csv` extension) or a GAUSS data file (`.fmt`). ASCII files must be delimited with spaces, commas, tabs or newlines. If your data is in an EXCEL file or can be put into an EXCEL file, you can save it as a tab delimited file (`*.txt`) or a space delimited file (`*.prn`) or a comma delimited file (`*.csv`) .[3] If no dimensions are specified, like in

```
load x = mydata;
```

then `x` is a column vector containing all of the data in mydata in the order 1st row, 2nd row, etc. (row major order). You can use the function `rows(x)`

[3]In the latest version of the software, it is now possible to load EXCEL files of the `.xls` type directly.

to find out if all the data have been loaded and the function `reshape (x, n, k)` to reshape the $nk \times 1$ vector `y` into the matrix you want to work with.

Once a data set is loaded into a matrix, you can go on and perform any kind of calculations you want on this `x` matrix or on some of its elements. It is now possible to create from this `x` matrix a GAUSS data set using the `SAVED` instruction

```
call saved(x,"mydata",0);
```

The resulting data set is called `mydata.dat`. These GAUSS data sets consist of two files, one with a `.dat` extension and another with a `.dht` extension. The latter is mainly an information file on the content of the corresponding `.dat` file. The last argument of the saved command is a $k \times 1$ string or character vector. If it is set equal to 0 like in this example, it simply means that the variables names will start with the character `X` and be followed by a number between 1 and k. In our example these will thus be `X1`, `X2`, `X3`.

An alternative would be to create this $k \times 1$ vector using `LET`

```
let names = var1 var2 var3;
@ creates a string vector "names" with 3 elements @
```

and then create a GAUSS data set using

```
call saved(x,"mydata",names);
```

When a GAUSS dataset exists on the disk (generated from a previous operation) you can open it directly using `OPEN` or `LOADD`. Another possibility is to load it directly into a matrix `x` with the line

```
x = "mydata";
```

5 Creating an output file

To create an output file, you need to direct the output of `PRINT` statements not only to the screen (this is automatic) but also to disk file. This is done using the command `OUTPUT`. It allows for path names. Example:

```
output file = filename on
```

where you can replace `on` by

```
reset or off
```

Without `on`, the command `OUTPUT` will only select the file to be used for the output but will not open it. A subsequent

```
output on; or output reset;
```

would then be required. The difference between `RESET` and `ON` is simply that in the latter case the file will be opened for appending, i.e. the results of the print statements are appended to the selected file if it already exists. In the `RESET` case a new file is created so that if the file already exists it will be destroyed. If you want to edit your output file with the GAUSS-edit window, you will first have to close the file using

```
output off;
```

6 The Basic operations with matrices

GAUSS basically works with matrices, and it is therefore important to note that almost all matrix expressions are entered following the usual way matrix expressions are written. Matrices are 2-dimensional arrays of double precision numbers which are all implicitly complex in the GAUSSI version (conversion between complex and real matrices occurs automatically in GAUSS). The matrices are stored in *row major order.* For example a 3×3 matrix will be stored in the following order:

[1,1][1,2][1,3][2,1][2,2][2,3][3,1][3,2][3,3]

Any matrix is indexed with 2 indices, vectors can be indexed with one index and scalars are considered as 1×1 matrices. The majority of functions and operators in GAUSS take matrices as arguments. Here are some useful ones when defining, saving and loading matrices:

=	assignement statement
|	vertical concatenation
~	horizontal concatenation
`LET`	matrix definition statement
`DECLARE`	is similar to the LET statement but for compile time matrices

The command `LET` thus creates matrices. The format for creating a matrix called `matrix` can take the following form:

Table 1: Examples of LET

	Shape of x
`let x = 1 2 3 4 5 6;`	Column vector 6x1
`let x = 1,2,3,4,5,6;`	Column vector 6x1
`let x = {1 2 3 4 5 6};`	Row vector 1x6
`let x = {1 2,3 4,5 6};`	Matrix 3x2
`let x[3,2] = 1 2 3 4 5 6;`	Matrix 3x2
`let x[3,2] = 1, 2, 3, 4, 5, 6;`	Matrix 3x2
`let x[3, 2] = 5;`	Matrix 3x2

`let matrix` = {*constant-list*};
or
`let matrix[r,c]` = {*constant-list*};

In the first case, the type of matrix depends on how the constants were specified. A list of constants separated by space will create a column vector. If, however, the list of constants is enclosed in braces {}, then a row vector will be produced. When braces are used, inserting commas in the list of constants instructs GAUSS to form a matrix, breaking the rows at the commas. If curly braces are not used, then adding commas has no effect. In the first case, the actual word LET is optional. If the second form is used, then an $r \times c$ matrix will be created; the constants will be allocated to the matrix on a row-by-row basis. If only one constant is entered, then the whole matrix will be filled with that number. Note the square brackets. This is the standard way to specify either the dimensions of a matrix or the coordinates of a block, depending on context. The first number always refers to the row, the second to the column.

Note that an assignment statement followed by data enclosed in braces is an implicit LET statement. Consequently, the following are equivalent:

```
let x = { 1 2 3, 4 5 6 };
x = {1 2 3, 4 5 6};
let x[2,3] = 1 2 3 4 5 6;
```

and create a 2×3 matrix x

$$\mathtt{x} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

When braces are used in LET statements, the commas define the row separation. A 2×3 matrix of ones is created with

```
let x[2,3] = 1;
```

or by using the command `ONES` which creates automatically matrices of ones:

```
x = ones(2,3);
```

while

```
let[2,3];
```

will create a 2×3 matrix of elements equal to 0. Note that the `LET` command cannot be used to define matrices in terms of expressions. The elements of the matrices, of rows or columns are easily isolated. For example, if we consider the `x` matrix given above

```
z = x[2,2];
```

will define `z` as a scalar and assign the value 5 to `z`, i.e. the value of the element of the second row, second column of the matrix `x`. To create a row-vector `x1` consisting of the 1st row of `x`, just use

```
x1 = x[1,.];
```

where the "." indicates "all columns". The resulting `x1` is a 1×3 vector

$$x1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

Once a matrix is created (or loaded), simple operations for matrix description and manipulations are easily performed like returning the number of columns, taking the max, sorting,... Here are a few of these standard manipulations often used when writing more complicated programs or procedures (for more details or details on the other manipulations see the manual or the on-line help):

`COLS(x)`	returns number of columns in the matrix `x`
`ROWS(x)`	returns number of rows in the matrix `x`
`MAXC(x)`	returns largest element in each column of the matrix `x`
`MAXINDC(x)`	returns row number of largest element in each column of the matrix `x`
`MINC(x)`	returns smallest element in each column of the matrix `x`
`MININDC(x)`	returns row number of smallest element in each column of the matrix `x`
`SUMC(x)`	computes the sum of each column of the matrix `x`
`CUMSUMC(x)`	computes cumulative sums of each column of the matrix `x`
`PRODC(x)`	computes the product of each column of the matrix `x`
`MEANC(x)`	computes mean value of every column of the matrix `x`
`STDC (x)`	computes standard deviation of every column of the matrix `x`

It is also often useful when you have for example a large matrix `x`, of dimension let say 150×35, to be able to consider sub-matrices of lower dimension consisting for example of a subset of the columns and the rows satisfying some criteria. This type of *extraction* is easily performed in GAUSS. There are various possibilities. Some of the most useful ones are:

- `PACKR(x)` that deletes the rows of a matrix that contain any missing values, "."., (Notice that the GAUSS code for a missing value is "." so that you have to replace whatever other code is used in your data before you import your data, although GAUSS has an internal procedure for conversion.
- `DIAG(x)` creates a column vector of dimension $min(n, k) \times 1$ vector from the diagonal of a $n \times k$ matrix. The matrix `x` need not be square.
- `LOWMAT(x)` returns the lower portion of a matrix, i.e. the main diagonal and every element below.
- `UPMAT(x)` returns the main diagonal and every element above.
- Elements of a matrix and submatrices can also be extracted using the row and column indices. For example `x[r1:r2,c1:c2]` extracts the submatrix consisting of rows $r1$ to $r2$ and columns $c1$ to $c2$. Using a dot, ".", in place of an index, extracts all the row or the column elements.

6.1 Precedence order of operators

Expression will be composed of operations on matrices. These are created using operators and the order in which the expression is evaluated is determined by the *precedence order of the operators* and the order in which they are used in the expression. As usual the multiplication and division operators (respectively * and /) have an higher order of precedence than the summation and substraction operators (+ and -).

6.2 Some matrix operators

The following mathematical operators work basically on matrices. Most of these assume numeric data. We list some of the frequently used operators with a few examples where we assume that we have already loaded two different matrices x and z, both of dimension $n \times n$.

Operators	*Code*	Examples
Addition	+	`y = x + z;`
Substraction	−	`y = x - z;`
Matrix Multiplication	*	`y = x * z ;`
Division or linear equation solution	/	`z = b/A;` where A and b are scalars
Element by element multiplication	.*	`y = x .* z;`
Element by element exponentiation	.^	`y = x .^z;`
Kronecker Product	.*.	`y = x .*.  z;`
Horizontal direct product	*~	`y = x *~y;`
Transpose operator	'	`y = x';`
Vertical concatenation	|	`y = x | z;`
Horizontal concatenation	~	`y = x ~y;`

Other operators/commands are given and detailed in the *Command Reference* manual. An often used operation in statistics and econometrics is the inversion of a given (invertible) matrix. This is most easily done using the command `INV` (or `INVPD` is the matrix is symmetric positive definite).

```
y = inv(x);
```

computes $y = x^{-1}$. The input of `INV` is thus simply the matrix you want to invert and the output is the inverted matrix. The determinant of `x` is obtained using `DET(x)`.

6.3 Comparing matrices, vectors and scalars

It is often very useful to compare matrices, scalars or vectors. These tasks are performed with *relational* operators and as usually with these type of operators, they return a scalar which we denote here by `z` which is either 0 or 1.

- Less than: `z = x < y;` or `z = x lt y;`
- Not equal: `z = x /= y` or `z = x ne y`;
- Greater than: `z = x > y; z = x gt y;`
- Greater than or equal to: `z = x >= y;` or `z = x ge y;`
- Equal to: `z = x == y;` or `z = x eq y;`
- Less than or equal to: `z = x <= y;` or `z = x le y;`

The result of all these operators is a scalar 1 (TRUE) or 0 (FALSE), based upon a comparison of *all elements* of `x` and `y`. Note that *all* comparisons must be true for a result of 1. It makes no difference whether you use the alphabetic or the symbolic form of the operator; however, in the alphabetic form the operator must be preceded and followed by a space. If you work with conformable matrices and want an element by element comparison which will return a matrix of zeros and ones, precede the operator a dot ".", e.g., .== or .eq.

7 Examples

At this point of the course, it is useful to take a look at some practical examples. First we focus on the linear regression and present different methods for dealing with the estimation of a linear regression problem. In a second step, we detail how to implement Monte Carlo simulations in GAUSS.

7.1 Linear regression

7.1.1 Linear regression: a brief review

Notation:

- Dependent variable: Y

- Number of observations: T

- k independent variables: X

Thus, Y has T rows and 1 column, X has T rows and k columns.
In matrix form, the OLS equation can be written as:

$$Y = X\beta + \epsilon \tag{1}$$

where ϵ (error term) is $N(0, \sigma^2)$. The OLS estimator is:

$$b = \widehat{\beta} = (X'X)^{-1}X'Y \tag{2}$$

The residuals are:

$$e = Y - \widehat{Y} \tag{3}$$

The unbiased estimator of the variance σ^2 is:

$$s^2 = \frac{e'e}{T-k} \tag{4}$$

The variance-covariance matrix of the estimated coefficients is:

$$V(b) = s^2(X'X)^{-1} \tag{5}$$

Student tests are performed using:

$$t_i = \frac{b_i - \beta_{i0}}{SE_i} \sim t(T-k) \tag{6}$$

where SE_i is the standard error of coefficient i.

7.1.2 OLS procedure

A first possibility is to use the `OLS` procedure provided in GAUSS. The dataset is included in the program.

```
/* LINEAR REGRESSION
We regress the consumption on income (US data, years 40-50) */

Z={
244 229.9,
277.9 243.6,
317.5 241.1,
332.1 248.2,
343.6 255.2,
338.1 270.9,
332.7 301,
318.8 305.8,
335.8 312.2,
336.8 319.3,
362.8 337.3
};

print "Number of observations " rows(Z);
print "Mean of Z " meanc(Z);

income=Z[.,1];
consum=Z[.,2];

{ vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat } = ols(0,consum,income);
```

See ols1.prg.

7.1.3 Matrix computation

A second possibility is to program the given formulae using the matrix operations in GAUSS.

```
income=Z[.,1];
consum=Z[.,2];

x=ones(rows(income),1)~income;
y=consum;

xxi=invpd(x'x);
b=xxi*(x'y);
e=y-x*b;
s2=e'*e/(rows(x)-cols(x));
sd=sqrt(diag(s2*xxi));
t=b./sd;

print "Estimated coefficients " b;
print "Standard errors " sd;
print "t-stats " t;
```

See ols2.prg.

7.1.4 Loading a data file

In the preceding programs, the data were included in the program. Usually, the data are given in an external file. In the following program, the data are loaded from an external text file.

```
n_obs=11;
load Z[n_obs,2]=c:\maas\qmif\progs\consinc.txt;
income=Z[.,1];
consum=Z[.,2];

x=ones(rows(income),1)~income;
y=consum;

xxi=invpd(x'x);
b=xxi*(x'y);
```

```
e=y-x*b;
s2=e'*e/(rows(x)-cols(x));
sd=sqrt(diag(s2*xxi));
t=b./sd;

print "Estimated coefficients " b;
print "Standard errors " sd;
print "t-stats " t;
```

See ols6.prg.

7.1.5 Loading a file and managing datasets

Finally, we illustrate the use of datasets. In the following program, the text file with the data is loaded and then written as a GAUSS dataset. This dataset is then used in the `OLS` procedure.

```
n_obs=11;
load Z[n_obs,2]=c:\maas\qmif\progs\consinc.txt;
income=Z[.,1];
consum=Z[.,2];

/* Dataset Z is written on disk */
dataset="c:\\maas\\qmif\\progs\\CONSINC";
let vnames=INCOME CONSUM;
if not saved(Z,dataset,vnames);
    errorlog "Write error";
endif;

dataset="c:\\maas\\qmif\\progs\\CONSINC";
let depvar=CONSUM;
let indvars=INCOME;
{ vnam,m,b,stb,vc,stderr,sigma,cx,rsq,resid,dwstat }
    = ols(dataset,depvar,indvars);
```

See ols5.prg.

7.2 Monte Carlo simulations

Monte Carlo simulations is a powerful technique used in applied econometric analysis. The GAUSS programming language is well suited for this task as it is fast and matrix operations are readily available. We consider a familiar application in quantitative finance, the diffusion process.

7.2.1 Diffusion process

The basic diffusion equation (with drift μ and volatility σ) can be written as:

$$dX_t = \mu X_t dt + \sigma X_t dW_t \tag{7}$$

with dW_t being a Wiener process, i.e. $dW_t = \epsilon_t \sqrt{\Delta t}$ and ϵ_t is IID N(0,1); $X(t_0) = X_0$. Note: dX_t should be understood as the change in X_t over the small time interval Δt. For this reason, we often write $\Delta t = h$ and $X_{t+h} - X_t = dX_t = \mu X_t h + \sigma X_t \epsilon_t \sqrt{h}$.

```
/* Start date */
t0=0;
/* End date */
tt=2;
/* Number of points */
N=200;
/* Number of simulations */
Ns=10;

/* Mu (year) */
mu=0.07;
/* Sigma (year) */
sigma=0.2;
/* Start price */
X0=50;

/* Step */
h = (TT-t0)/N;
/* Sequence of time increments */
t = seqa(t0,h,N+1);

xt = zeros(N+1,Ns);
xt[1,.] = x0*ones(1,Ns);
u = rndn(N,Ns)*sqrt(h);

i = 1;
do until i > N;
    xi = xt[i,.];
    xt[i+1,.] = xi + xi*mu*h + xi.*(sigma*u[i,.]);
    i = i + 1;
```

```
endo;
```

See mcdp1.prg.

8 Graphics in Gauss

One attractive feature of GAUSS is its ability to draw high quality graphics in two of three dimensions of a large variety of different types. GAUSS also provides a complete windowing system for plotting multiple graphs on one page. The way GAUSS works is to provide functions which draw the graphs and only draw the graphs. All other attributes are set using (global) variables. Hence, the creation of a graph involves setting one variable to the title, another to the type of lines wanted, another to the color scheme,... When the graph function is then called, GAUSS uses all the information previously set to draw the graph with the right characteristics. Any program drawing graphs should have the line

```
library pgraph;
```

ideally at the start of the program. This enables GAUSS to know where all the specialized graph-drawing routines are to be found. Remark that graphs cannot be drawn if this line is omitted. The `LIBRARY` line should appear only once at the top of your program and could be followed by `GRAPHSET` which reset all the variables back to their default values.

There are an enormous amount of options that may be specified, all these are detailed and specified in the *System and Graphics Manual*, and some information on these can be obtained from the on-line help. They all begin with `_p` to make them easily to identify. These are set/modified like any other variables in GAUSS. For example,

```
_pcolor = zeros(2,1);

_pcolor[1] = 2;

_pcolor[2] = 5;

_pbartyp = {2 1, 2 2, 2 3};
```

The `_pcolor` instruction sets colors for the XY and XYZ graphs. It is a 2x1 vector implying, in this case, that there are two series to be plotted. The first

series will be plotted in the color defined by 2 (which is grey), the second in red.
The `_pbartype` instruction sets the shading type and color for a bar graph. It is a 3x2 matrix, implying three series. The most useful variable is

```
_plegstr = "legend A\000legend B\000Legend C";
```

This defines legends for each line when a graph is displaying multiple series - three in this case. The legends for each series must be separated by the code

```
\000
```

which is a null character specifying that one name has ended and another is beginning. The relevant variables to be set are detailed with each graph type. In addition there are a number of general functions which control other settings, of which the most important are

```
title(title);

xtics(min, max, increment, subDivs);

xlabel(title);
```

The first of these sets the title for the graph. `XTICS` (and the associated functions `YTICS` and `ZTICS`) allow for scaling of the X-axis. If this function is not called, GAUSS will work out its own scaling. `min` and `max` are the minimum and maximum values on the scale, with the scale increasing by increment; negative values for the increment are acceptable. `subDivs` is the number of minor ticks between each increment. Finally, `XLABEL` (and `YLABEL` and `ZLABEL`) provides a title for the X, y or Z axis.

All these options should be set before printing a graph. However, most of the defaults are good choices, and many options will not need changing. GAUSS provides a number of graph types, most importantly bar graphs, X-Y, log X-Y and histograms. All data for graphs come in the form of matrices. When a graph instruction is encountered, GAUSS plots the graph immediately using the current set of options or defaults. This is why all the options are set first. By the time GAUSS reaches a graph instruction, all it needs to produce the graph is the data given in the function call. The graph data are in $N \times K$ matrices, where N is the number of data points and K is the number of series to be plotted. Whether multiple series are permitted or not depends on the graph: for example, multiple series are allowed in an X-Y graph. For example

```
xy(var1, var2~var3);
```

will plot an X-Y graph, using a Cartesian coordinate system, consisting three series where `var1` will specify the x-axis. `LOGX` (resp. `LOGY`) will plot an X-Y graph using logarithmic X (resp. Y) axis while `LOGLOG` will plot an X-Y graph using logarithmic X and Y axes.

In practical data analyses, it is often useful to have different graphics on the same page. This is easily done by using the complete windowing system for plotting multiple graphs on one page provided in GAUSS. We will just mention here the most important steps that one has to follow. In all cases, one has to initialize the window procedure with `begwind` that *must* be called before any other window functions are called. Similarly `endwind` ends the window manipulation and it is only thereafter that the graphs are displayed. An important procedure is `window` which creates and partitions the screen into windows of equal size. The general format is `window(#row,#col,typ)`; where #`row` is the number of rows of windows and #`col` the number of columns of windows while `typ` specifies the window attribute type: if this value is 1, the windows will be transparent, if 0, the windows will be non-transparent. Notice that the windows will be numbered from 1 to (#*row* × #*col*) beginning from the left topmost window and moving to the right. The current window is set to 1 immediately after calling this function, `setwind` is used to specify the window number. One may alternatively use nextwind which sets the current window to the next available window number.

Graphs (or pages with several graphs) are automatically displayed in specific windows and can be printed, saved or converted into various formats that may later be used in other packages (such as word processors like Winword, WP, Scientific Word,...). You can print the graphs that GAUSS generates directly, but if you want to put them into a wordprocessor GAUSS supports conversion to four formats: `ps` (postscript), `pic` (Lotus), `hpgl` (plotter) and `pcx` (bitmap).

8.1 Example

The following lines illustrate this idea of putting various graphics on the same page. We first generate some random numbers, accumulate these to generate a random walk. Lagged and first differences are generated. Finally a page with four graphs is created.

```
library pgraph; graphset;

n = 400; e = rndn(n,1); /* generating pseudo (normal) random data */
```

```
obser = seqa(1,1,n); /* generate a deterministic series 1,2,...,n */

y = CUMSUMC(e); /* cumulate the y series to get a random walk */
yl = lagn(y,1); /* generate lagged variable */
yd = y-yl; /* generate first differenced series */

begwind; window(2,2,1);
setwind(1); title("Figure 1");
xlabel("Observations"); ylabel("Variable y - level");
xy(obser,y);
setwind(2); title("Figure 2");
xlabel("Observations"); ylabel("Variable y - lag");
xy(obser[1:rows(yl),.],yl);
setwind(3); title("Figure 3");
xlabel("Observations"); ylabel("Variable y - 1st diff.");
xy(obser[1:rows(yd),.],yd);
setwind(4); title("Figure 4");
xlabel("Random numbers"); ylabel("Frequencies.");
{b1,m1,freq1} = HISTP(y[.,1],50);
endwind;
```

See graphs1.prg.

9 Loops, conditional branching...

9.1 Conditional branching

The syntax of the full `IF` statement is:

```
if condition1;
    dothis1;
elseif condition2;
    dothis2;
elseif condition3;
else;
    dothis4;
endif;
```

but all the `ELSEIF` and `ELSE` statements are optional. Thus the simplest `IF` statement is

```
if condition1;
    dothis1;
endif;
```

Each condition has an associated set of actions (the `dothis`). Each condition is tested in the order in which they appear in the program; if the condition is "true", the set of actions will be carried out. Once the actions associated with that condition have been carried out, and no others, GAUSS will jump to the end of the conditional branch code and continue execution from there. Thus GAUSS will only execute one set of actions at most. If several conditions are "true", then GAUSS will act on the first true condition found and ignore the rest. If none of the conditions is met, then no action is taken, unless there is an `ELSE` part to the statement. The `ELSE` section has no associated condition; when the `ELSE` statement is reached, GAUSS will always execute the `ELSE` section. To reach the `ELSE`, GAUSS must have found all other conditions "false".[4]

9.2 Loop statements: WHILE and UNTIL, FOR

The format for the loop statements are

```
do while condition;                 do until condition;
   dothis;                             dothis;
endo;                               endo;
```

These two are identical except that the first loops until condition is "false", while the second loops until condition is "true". This means that

```
do while  condition;          do until (NOT condition);
```

are identical. The operation of the `WHILE` loop is as follows: (i) test the condition; (ii) if "true", carry out the actions in the loop; then return to stage (i) and repeat; (iii) if "false", skip the loop actions and continue execution from the first instruction after the loop. Note that the condition is tested before the loop is entered. Also, there is nothing in the definition of the loop to say how the loop condition is set or altered. It is the programmer's responsibility to ensure that the condition is set properly at each stage. Since version 3.2.35, GAUSS now also offers a `FOR` loop construct. The format is

```
for i (start, stop, step);
    dothis;
endfor;
```

[4]Unconditional branching is done with the `GOTO`. The target of a `GOTO` is called a `label`. Labels must begin with '_' or an alphabetic character and are always followed by a colon.

where `i` stands for the counter variable, `start` (may be a scalar expression) is the initial value of the counter, `stop` is the final value of the counter and `step` the increment value. Note that the counter is strictly *local* to the loop. The commands `BREAK` and `CONTINUE` are supported. The `CONTINUE` command steps the counter and jumps to the top of the loop. The `BREAK` command terminates the current loop by jumping past the `ENDFOR` statement. When the loop terminates, the value of the counter is stop if the loop terminated naturally. If break is used to terminate the loop and you want the final value of the counter you need to assign it to a variable before the break statement.

9.2.1 Examples

```
x = zeros(10, 5);
for i (1, rows(x), 1);
    for j (1, cols(x), 1);
        x[i,j] = i*j;
    endfor;
endfor;
```

Example using `BREAK`:

```
li = 0;
x = rndn(100,1);
y = rndn(100,1);
for i (1, rows(x), 1);
    if x[i] /= y[i];
        li = i;
        break;
     endif;
endfor;
if li;
    print "Compare failed on row " li;
endif;
```

10 Defining and using procedures

As already mentioned above you should as much as possible make use of procedures which allow you to define a new function that you create and which can then be used later as if is was an *intrinsic* function. Procedures are extremely useful once you are confronted with an excessively large and complicated program that may be difficult to read, understand, and alter.

If the program is broken into separate sections with meaningful procedure names, it becomes much more manageable. Alternatively, there may be a piece of code which carries out some minor function. Placing this code in a procedure allows the programmer to concentrate on the main points of the program. The second most important reason to motivate the use and the construction of procedures is the repetitive character of numerous operations. The choice is then simply between explicitly programming the same operation several times, or writing a procedure and calling it several times; usually the latter wins hands down. Finally, procedures are often easier to test and less susceptible to unexpected influences.

Hence procedures are extremely useful when the same estimation method, test statistic, expression or model evaluation is to be used often in quite different situations. A procedure is thus *a user defined function* that is later used as if it was an intrinsic part of the GAUSS language. Notice that any intrinsic command of function, and any user defined function of procedure can be used *within* a procedure.
A procedure definition consists of 5 different parts:

1. Procedure declaration (`PROC` Statement): the format of the `PROC` statement is as follows:

   ```
   PROC ({number of returns}) = name(arguments);
   ```

   The *arguments* can be numerous and are then separated by commas. These are the names that are used inside the procedure for the arguments that are passed to the procedure when the latter is called.

2. Local variable declaration (`LOCAL` statement): local variables are only known within the procedure defined. Remark that there is no information about the size or type of the local variable here. All this statement says is that there are variables which will be accessed during this procedure, and that GAUSS should add their names to the list of valid names while this procedure is running.

3. Body of the procedure: the body contains GAUSS statements that are used to perform the task, they may use intrinsic functions, used defined functions,

4. Returns from the procedure (`RETP` statement): returns a number of items. Their number must correspond with the *number of returns* in the `PROC` statement. These returns can however be of any type. It is important to know that GAUSS will not check these returns and it will

not warn the user if the number of returns is not equal to the number of returns specified in the procedure declaration. GAUSS will only report an error when the procedure is actually called during a program run.

5. End of procedure (`ENDP` statement): the statement `ENDP` tells GAUSS that the definition of the procedure is finished. GAUSS then adds the procedure to its list of symbols. It does not do anything with the code, because a procedure does not, in itself, generate any executable code. A procedure only exists when it is called; otherwise it is just a definition.

There are different ways of calling a procedure. Assume that the procedure's name is given by `procname` and the arguments are `i, j and k.` Procedures are called as follows

```
call procname(i,j,k); /* will ignore all returns */
y = procname(i,j,k); /* is suited for one return procedures */
{ x, y, z \} = procname(i,j,k); /* is suited for multiples returns*/
procname(i,j,k); /* no returns */
```

10.1 Example

Let us here give the example of a procedure that estimates a linear regression model and returns the OLS estimates, their standard errors and t-values (see the formulae given above).

```
income=Z[.,1];
consum=Z[.,2];

x=ones(rows(income),1)~income;
y=consum;

{b,sd,t}=regress(x,y);

print "Estimated coefficients " b;
print "Standard errors " sd;
print "t-stats " t;

proc (3)=regress(x,y);
local xxi,b,e,s2,sd,t;
xxi=invpd(x'x);
b=xxi*(x'y);
e=y-x*b;
```

```
s2=e'*e/(rows(x)-cols(x));
sd=sqrt(diag(s2*xxi));
t=b./sd;
retp(b,sd,t);
endp;
```

See ols3.prg.

The different lines have the following interpretation:

- First the procedure declaration defines that the procedure will be called `regress` and will return `3` matrices.
- The names `x` and `y` will be used inside the procedure. These are the (*arguments)* names that are used inside the `regols` procedure for the arguments that are passed to the procedure when `regols` is called.
- We then define the local variables.
- The next six lines form the body of the procedure: we first calculate $(x'x)^{-1}$ (denoted by `xxi`) use it to compute `b` and to calculate the ols residuals `e`. An estimate of the error variance is computed (`s2`) and used to obtain the standard error of b (`sd`). Finally the `t`-values are calculated. This ends the body of the procedure.
- `RETP` then returns the three vectors `b, sd, t`: the OLS estimates, their standard deviations and the corresponding t-values. `ENDP` closes the procedure.

11 GAUSS modules

The basic GAUSS program is the first building block of the GAUSS software package. Other program packages (modules) are available from *Aptech Systems* and focus on more specific applications such as maximum likelihood estimation, time series methods or financial applications. In this introductory text, we briefly review an application of the maximum likelihood module and the time series module (ARIMA modelling).

11.1 Maximum likelihood

11.1.1 A brief review of maximum likelihood estimation

- Model parameters: θ. The true values of the parameters are θ_0, which are unknown.

- Number of observations: T

- Observations: y_t, $t = 1 \dots T$, and possibly explanatory variables x_t (y and x vectors in matrix form).

- Density function for the model: $f(y, \theta)$

- Likelihood function based on the density function: $L(\theta, y) = f(y, \theta)$. For a given set of observations y, $L(\theta', y)$ gives the likelihood that the sample y has been generated by the density law $f(y, \theta')$. If we have that $L(\theta', y) > L(\theta'', y)$, then it is more likely that the sample y is a realization of $f(y, \theta')$ than $f(y, \theta'')$. In other words, θ' is a better candidate for θ_0 than θ''.

- The maximum likelihood estimation procedure is the following optimization problem:
$$max_\theta L(\theta, y) \Longrightarrow \widehat{\theta} \tag{8}$$

- Based on $\widehat{\theta}$, the maximized likelihood is $L(\widehat{\theta}, y)$.

- In general, one maximizes the log-Likelihood, or $ln(L(\theta, y)) = l(\theta, y)$.

- Score or gradient of the log-Likelihood: $s(\theta, y) = \partial l(\theta, y)/\partial\theta$. By definition of the maximization problem, $\widehat{\theta}$ is such that $s(\widehat{\theta}, y) = 0$, which allows the computation of $\widehat{\theta}$.

- Hessian matrix: $H(\theta, y) = \partial^2 l(\theta, y)/\partial\theta\partial\theta'$

- Information matrix: $I(\theta) = -E(H(\theta, y)) = E(s(\theta, y)s'(\theta, y))$

- Asymptotic information matrix: $IA(\theta) = lim_{T\to\infty} I(\theta)/T$

11.1.2 Two important properties of the maximum likelihood estimator

- The ML estimator is asymptotically unbiased and fully efficient.

-
$$T^{1/2}(\widehat{\theta} - \theta_0) \to N_k(0, IA^{-1}(\theta_0)) \tag{9}$$

  In practice, θ_0 is unknown and can be replaced by $\widehat{\theta}$. Because $IA(\widehat{\theta})$ can be difficult to compute, one can replace it by $-H(\widehat{\theta}, y)/T$. This leads to

$$\widehat{\theta} - \theta_0 \to N_k(0, (-H(\widehat{\theta}, y))^{-1}) \tag{10}$$

The variance-covariance matrix can be computed using several methods (see below).

11.1.3 Numerical procedures

- Iterative numerical procedure for θ: $\theta_1, \theta_2, \ldots, \theta_N$ such that $l(\theta, y)$ is maximized when $\theta = \theta_N$.
- Stepsize: λ_i
- Direction: d_i
- Iterative procedure for θ: $\theta_{i+1} = \theta_i + \lambda_i d_i$
- Because $l(\theta_i + \lambda_i d_i) = l(\theta_i) + \lambda_i s'(\theta_i) d_i$ using a first order approximation, an immediate choice for d_i is $d_i = s(\theta_i)$ (and $\lambda_i > 0$). More generally, one has $d_i = Q_i s(\theta_i)$, where Q_i is a symmetric positive definite matrix.
- Numerical procedures thus involve a choice for λ_i and most importantly the specification of d_i and Q_i. Some examples for Q_i: (1) $Q_i = I$, steepest descent; (2) $Q_i = -H_i^{-1}$, Newton method; (3) $Q_i = (s(\theta_i)s'(\theta_i))^{-1}$, BHHH procedure.
- Stop rule: the numerical procedure usually stops when a convergence threshold has been reached, i.e. $|l(\theta_N) - l(\theta_{N+1})| < \eta$, where η is the given threshold.
- Variance-covariance matrix of θ: as indicated above, one can use $V(\theta) = (-H(\theta_N))^{-1}$ or $V(\theta) = (s(\theta_N)s'(\theta_N))^{-1} = (\sum_{t=1}^{T} s_t(\theta_N)s_t(\theta_N)')^{-1}$, where $s_t(\theta_N) = \partial l(\theta = \theta_N, y_t)/\partial\theta$ (BHHH method). Another possibility is to compute the QML variance-covariance matrix.

11.1.4 Gauss example

The following program provides an example of maximum likelihood estimation for the ordinary least squares problem studied earlier. The likelihood is based on the normal distribution. As the maximum likelihood module features a whole range of estimation options, we also provide a brief description of the most important global variables which determine the outcome of the estimation process. Further information can be found in the maximum likelihood *Reference* manual and in the comments of the `maxlik.src` procedure.
Remark: QML estimation is directly available by selecting _max_CovPar=3.

```
/* Maximum likelihood estimation */
library maxlik;
maxset;

/* Simulation of the data */
nobs=1000;
beta=1 | -1 | 0.5 | -0.25;
s=2;
x=ones(nobs,1)~3*rndn(nobs,3);
y=x*beta+s*rndn(nobs,1);

/* Maximum likelihood estimation */
/* Vector of starting values */
start_val=ones(5,1);

/* Names of parameters
**  _max_ParNames - Kx1 character vector, parameter labels.
*/
_max_ParNames="BETA1" | "BETA2" | "BETA3" | "BETA4" | "S";

/* Optimization algorithm
**  _max_Algorithm  -  scalar, indicator for optimization method:
**              = 1,   SD (steepest descent)
**              = 2,   BFGS (Broyden, Fletcher, Goldfarb, Shanno)
**              = 3,   DFP (Davidon, Fletcher, Powell)
**              = 4,   NEWTON (Newton-Raphson)
**              = 5,   BHHH
**              = 6,   Polak-Ribiere Conjugate Gradient
*/
_max_Algorithm =5;

/* Step length
**  _max_LineSearch - scalar, indicator determining the line search method.
**
**            = 1,  steplength = 1
**            = 2,  STEPBT  (default)
**            = 3,  HALF
**            = 4,  BRENT
**            = 5,  BHHHSTEP
**
**            Usually _max_Step = 2 will be best.  If the optimization
**              bogs down try setting _max_Step = 1 or 3.  _max_Step = 3
**              will generate slow iterations but faster convergence and
**              _max_Step = 1 will generate fast iterations but slower
**              convergence.
*/
_max_LineSearch =2;

/* Method for computing the variance-covariance matrix at the end of the optimiz
**  _max_CovPar  -  scalar, type of covariance matrix of parameters,
**               = 0,  the inverse of the final information matrix from
**                     the optimization is returned in cov (default).
**               = 1,  the inverse of the second derivatives is returned.
**
**               = 2,  the inverse of the cross-product of the first
**                     derivatives is returned.
**
**               = 3,  the hetereskedastic-consistent covariance matrix
**                     is returned.
*/
_max_CovPar=0;

/* Convergence criteria
**  _max_GradTol  - scalar, convergence tolerance for gradient of estimated
**              coefficients.  Default = 1e-5.  When this criterion has been
**              satisifed OPTMUM will exit the iterations.
*/
_max_GradTol =1e-5;

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

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;
```

See ml1.prg.

11.2 Time series and ARIMA models

Autoregressive Integrated Moving Average models are quite popular and often used in univariate time series analysis, especially for the purpose of forecasting. The general form of an ARIMA (p,1,q) is

$$\sum_{i=0}^{p} \phi_i \Delta x_{t-i} = c + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i}$$

where Δ is the first difference operator and ε is a white noise. Assume we want to analyze such a model for a time series x which we have already loaded in a $T \times 1$ vector `x`. The general library containing all procedures useful for univariate time series analysis is `arima.lcg` The top of the program should therefore start with

```
Library arima; /* activates the library ARIMA */
arimaset; /* reset the global variables to their default values */
```

Assume now that you first want to compute the sample autocorrelation function of the first differenced times series (i.e. consider $d = 1$). For this purpose you can simply use the `ACF` procedure which is a one return procedure and can therefore be called with

```
a = ACF(x,lagmax,d);
```

where `x` is the series, `lagmax` the maximum lag you want to consider and `d` the degree of the difference operator. This single return procedure will generate a $(lagmax \times 1)$ vector containing the sample autocorrelation for lag 1 to lagmax and call this vector `a`.

Consider now the estimation of the ARIMA(p,1,q) model and assume that you fix $p = q = 1$. Maximum likelihood estimates are obtained using the `arima` procedure which is a multiple return procedure that you can therefore call by

```
{coefs,e,ll,vcb,aic} = arima(startv,y,p,d,q,const);
```

Between braces you find the returned matrices: `coefs` are the MLE of the parameters $(\phi_{i,}\phi_{i,}\theta_{i,}c)$. `e` is the vector containing the fitted residuals, `ll` the value of the log-likelihood function, `vcb` the estimated variance covariance matrix and `aic` is the value of Akaike's information criterion.

The arguments of the procedure are given on the r.h.s. between parentheses: `startv` is a vector of starting values. If set to 0, the procedure calculate these automatically. `y` is the series analyzed, `p,d,q` are the order of the AR part, the order of differencing and the order of the MA polynomial and `const` specifies if the model contains a constant or not. As it is often the case with estimation procedures, you can change the default values of some options by modifying global variables. For example you can type

```
_iterol = 250;
```

which will increase the maximum number of iteration up to 250 (the default value of `_iterol` is 100).

12 Libraries

A Gauss library will serves as a sort of dictionary to the source files that contain the definition of the symbol, or the procedure, etc. The Gauss library is thus a *text file*, the extension must be `.lcg`.

A library file will contains several left flushed filenames with an extension `.src` (for source), `.dec` (for declaration) or `.ext` (external) and located in the SRC subdirectory (and/or the other subdirectory specified in the `Gauss.cfg` under src_path). These are the files where the symbols, procedures are defined, global variables declared for compilation purposes and external declarations. After this left flushed filename you must include the symbols included and defined in these files as well as their type (procedures, matrices). The library files are located in the subdirectory LIB (specified in the `Gauss.cfg` under lib_path). Let us here look at parts (much has been deleted) of the Linear Regression library file called `lr.lcg`:

```
/* lr.lcg - Linear Regression Library
(C) Copyright 1992-1994 by Aptech Systems, Inc.
All Rights Reserved. */

lr.dec

  lreghc  : matrix
  lregres : matrix

... (lines omitted)

l2sls.src

  l2sls   : proc
  lreg.src

... (continues)
```

Files with the `.dec` extension are files where you declare global variables using the `DECLARE` statement which initializes global matrices and string used by procedures in a library system.[5] It generates no executable code and is only for compile time initialization. In this part two global matrices are declared. `_lreghc` is in fact a scalar whose default value is 0. If, you set it to 1 by including in your program the line

```
_lregc = 1
```

then a heteroscedastic-consistent covariance matrix estimator will be calculated. Similarly `_lregres` is a string, a file name to request so-called influence diagnostics. The statistics generated from the diagnostics will then be saved under the given file name. Global variables are thus useful for options definition. Their defaults values are usually defined in a ...set file. In this case, the default values of the global variables used in `lr.lcg` are defined in `lrset.src` (in the SRC subdirectory).

Two lines below you find the left flushed filename `l2sls.src` which is the source file containing a procedure for Two Stages least Squares regressions.

For GAUSS to be able to look for the symbol within a given library, the latter must however be *activated.* The activation of given libraries is simply realized by including a `LIBRARY` statement at the top of the program. Example: by including the line

```
library lr; /* the .lcg extension is not required */
```

at the top of your program you will be able to use all the functions, procedures,... that are defined in the files that are part of the `Linear Regression Library`.

[5]In contrast to LOCAL variables, matrices,... global variables are known with their names both within and outside the procedure. These are also the variables you might want to change if you want change the default options of the procedures (when these options do exist).