

INTRODUCTION TO GAUSS - 4034T

Alain Hecq

Department of Quantitative Economics

Maastricht University

Room 2085..

`a.hecq@ke.unimaas.nl`

`http://www.personeel.unimaas.nl/a.hecq`

This version: January 2003

Contents

1	Introduction	3
2	Getting started: Command and Edit modes	5
3	General Overview: where we talk about variables, statements, programs, libraries and procedures	9
4	Loading data and creating data sets	13
5	The Basic operations with matrices	15
5.1	Some matrix operators	19
5.2	Comparing matrices, vectors, scalars and strings	21
5.3	Conditional branching	22
5.4	Dynamic modeling and creation of lags	23
6	Examples: Ordinary least squares estimator (OLS)	24
6.1	Linear regression: Notations	24
6.1.1	OLS Program	25

7	Some tricks when writing out programs: personal point of view	28
7.1	Ô Memory	28
7.2	Annotation and comments	28
7.3	Format	29
7.4	Creating an output file	29
7.5	The Con Stuff	29
8	Defining and using procedures	30
8.1	Example 1	32
8.2	Example 2	33
8.3	Example 3	33
8.4	The Functions using FN	34
9	Libraries	36
10	Graphics in Gauss	37
10.1	Example 1	39
10.2	Example 2	40
11	Loop statements: While, Until and For	41
12	Small-Sample Inference: Resampling Methods	44
12.1	Monte Carlo Simulations	44
12.2	Pseudo-random numbers and the use of the seed	49
12.3	Bootstrap	51
13	Gauss modules	52
13.1	Maximum likelihood	52
13.1.1	A brief review of maximum likelihood estimation	54
13.1.2	Two important properties of the maximum likelihood estimator	55
13.1.3	Numerical procedures	56
13.1.4	GAUSS example	56
13.2	Time Series and ARIMA Models	59

1 Introduction

This manuscript is a guide to using the software package GAUSS 3.2 for windows.¹ 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 to refer to 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. For those not having the possibility to consult these manuals, the on-line help system in GAUSS 3.2 is a valuable help to find the meaning and to remember the formulation of options of GAUSS commands. In order to make easier your contact with GAUSS I have copied in the Appendix its main commands. These come from the official GAUSS website at:

<http://www.aptech.com/weblist.pdf>

Copies of the 272 pages *User Guide* and the 950 pages *Reference* manual can be found at

<http://www.aptech.com>

(if you click on manuals icon) or some at some of the non-official addresses such that

http://www.caspur.it/risorse/softappl/doc/gauss_docs/UserGuide.pdf

http://www.caspur.it/risorse/softappl/doc/gauss_docs/LanguageReference.pdf

Notice that these manuals, while helpful, also include the new commands available in the new GAUSS 4.0 and 5.0 versions and not in the version you will work with.

It's important to note from the outset that GAUSS is a programming language such as FORTRAN or C and not a statistical or an econometric package such as RATS, TSP, SAS and even less a "ready to click" product such as SPSS, PCGIVE, MICROFIT or EVIEWS. However GAUSS is designed to operate with and on matrices and that makes econometric

¹GAUSS is a trademark of *Aptech Systems Inc.*, Maple Valley, USA. <http://www.aptech.com>. Although the GAUSS 5.0 version is now available, we do not have a site licence for that new release. Consequently we keep on working with GAUSS 3.2.35.

modeling very easy. For example, once you have entered a $T \times k$ matrix $\mathbf{X}$ and a $T \times 1$ vector $\mathbf{y}$, the statement `b=inv(X'X)*X'y` provides the least squares estimate of the linear relationship of $\mathbf{y}$ on $\mathbf{X}$. Obviously, at the first glance that is more complicated and more involving than just clicking with the mouse in EVIEWS but GAUSS gives an incredible flexibility enabling the user to go beyond the tools already pre-formatted for him in usual econometric software. Let us just quote the use of **loops**, the **computation of non-standard maximum likelihood functions** or **Monte Carlo experiments**. 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 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.

As far as money is not concerned (the new version is quite expensive and additional modules must be bought separately), there are at least **two main disadvantages** for using GAUSS.

- The price to pay for its flexibility is naturally that you often have to do the job on your own, namely to write down your own programs, procedures, estimators. **But** there are two things to balance the previous assertion.
 - First, GAUSS is so widely widespread that programs are likely available somewhere else and can be downloaded from the net. Have a look at the GAUSS part in the software section of:

<http://econometriclinks.com>

as well as at

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

to see the incredible source and number of program codes you can access.

- Second, additional modules are available for time series (ARIMA), maximum likelihood, optimization...
- The second drawback I see is that GAUSS is so flexible that it is sometimes too tolerant with sloppy programming which means that it is difficult for the computer

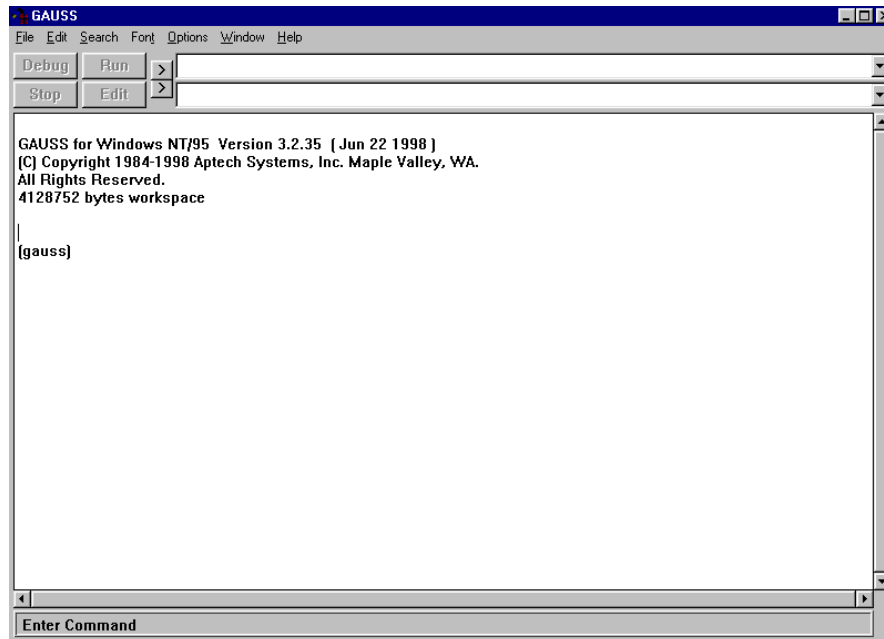


Figure 1: Screen of the Command mode

to tell when mistakes occur. A simple example is the estimation of a univariate autoregressive model of order one $y_t = \rho y_{t-1} + \varepsilon_t$. Indeed, if one defines $\mathbf{X}=\text{lag1}(\mathbf{y})$ it could happen that either you would not obtain any OLS estimate or you could obtain incorrect results because GAUSS is going to make the calculus on the T observations although you must discard the first one and work with the remaining $T - 1$ ones. This is usually automatically done in most econometric packages.

2 Getting started: Command and Edit modes

There are two different ways to run GAUSS, the *Command mode* and the *Edit mode*. When you start GAUSS under Windows you simply have to double-click on the GAUSS short cut icon and you are automatically set in *Command mode* whose screen is reproduced in Figure 1. The screen gives you the actual version of GAUSS as well as the amount of free workspace memory. Gauss commands are typed after the GAUSS prompt (`[gauss]`) and are executed when you press the `<Enter>` key. For example, in the following lines I first create a three dimensional identity matrix. Then I multiply it by the scalar 3 and I print the result.

```
(gauss) x=eye(3)    /* I create a 3 by 3 identity matrix */ ; <Enter>
(gauss) y=x*3;    @  I multiply x by 3 and call this matrix y @ <Enter>
(gauss) y <Enter>
      3.0000000 0.0000000 0.0000000
      0.0000000 3.0000000 0.0000000
      0.0000000 0.0000000 3.0000000
(gauss)
```

Notice there exist two ways for introducing comments that are not read by Gauss, namely:

```
@ ... @  and  /* ... */
```

Also remark that while the semicolon (;) is not necessary in the *Command mode* it is compulsory if you add information such as remarks or comments.

Menus of interest within the command window in Figure 1 are: **File, Edit, Search, Font, Options, Window, and Help**. We briefly describe their aims because their meaning should be obvious to you from the context.

- **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 file name 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 let 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.

But one can imagine that only small jobs can be done in this mode and the *Edit mode* is the more natural one. The *Edit mode* is simply a text editor where the program will be created. 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 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. If you have previously created a file, just go to **File** and then **Editor** in the bar menu, otherwise if you want to create a new file that you will call **myprog.pgm** simply type in the Command window after the GAUSS prompt.

```
(gauss) Edit myprog.pgm <Enter>
```

and then the GAUSS-edit window will automatically appear (see Figure 2). You are then in *Edit mode* and already in **myprog.pgm**. You can enter your data set or create new

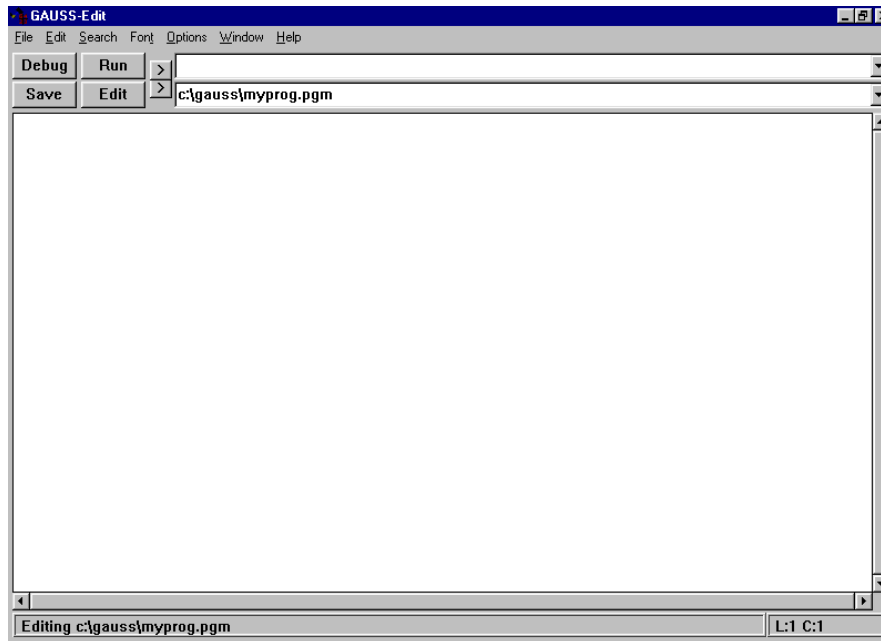


Figure 2: GAUSS-edit window

program as in any usual text editor. If you have already a saved copy of your program, let say `myprog.pgm`, on your (hard)disk, just type

```
(gauss) Edit myprog.pgm <Enter>
```

and you are then in *Edit mode* where you can modify your `myprog.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 should introduce a path name:

```
(gauss) Edit A:\myprog.pgm <Enter>
```

which specifies where to go to get `myprog.pgm`. 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.

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.

Remark that it could be convenient to create a new folder in Gauss to put data sets and programs relative to a project or a course. For instance or

```
(gauss) Edit c:\GAUSS\cours\myprog.pgm <Enter>
```

or simply by editing and creating a new file in that particular subdirectory.

3 General Overview: where we talk about variables, statements, programs, libraries and procedures

The central concept in programming with GAUSS is **variables**. There are two basic of such variables or data types supported in GAUSS: **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. Note that all variables must be created and given an initial.

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 `'='`.

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 (in Command mode):

```
(gauss)x=10; <Enter>
(gauss)y=x*3; <Enter>
```

are statements. The following two statements are equivalent for getting the value of `y`:

```
(gauss)print y; <Enter>
(gauss)y; <Enter>
```

Examples of data types

- Numerical 3×3 matrix

$$\begin{pmatrix} 1 & 0 & -3 \\ 6.2 & 9 * 10^{-6} & 5 \\ 7 & 9 & 99 \end{pmatrix}$$

- Character 2×2 matrix

$$\begin{pmatrix} \textit{Bush} & \textit{No} \\ \textit{Barbie} & \textit{Yes} \end{pmatrix}$$

- Strings

```
Var1='Strings may be pieces of text of long length'
```

```
Var2='''
```

```
var3='2.2'
```

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** depends on whether **var1** is a numerical matrix, a character matrix or a string.

```
(gauss)print var1; /* or only var1; */ <Enter>
(gauss)print $var1; /* or only $var1; */ <Enter>
```

Here are some examples making these distinctions I hope clearer.

```
(gauss) x=7 <Enter>
(gauss) x <Enter>
7.000000
(gauss) y='this is a string' <Enter>
(gauss) y <Enter>
this is a string
(gauss)
```

but

```

(gauss) let v = this_is_a_character <Enter>
(gauss) v <Enter>
1.5783153e+151 <Enter>
(gauss) $v <Enter>
THIS_IS_ <Enter>
(gauss)

```

With care, one can also make operation with strings or characters similar to the one applied to numerical data. For example, define two strings and concatenate them horizontally such as

```

(gauss) s= ''nic''~''ta mere''; <Enter>
(gauss) s; <Enter>
4.7709565e+180 2.6846834e-315
(gauss)

```

but

```

(gauss) s= ''nic''~''ta mere''; <Enter>
(gauss) $s; <Enter>
      nic      ta mere
(gauss)

```

As opposed to strings, remark the truncation (at 8 elements) of text in the character and mixed matrices. For instance (we'll see below how to create matrices):

```

(gauss) let x = aaaaaaaaaaaaaaaaaa bbbbbbbbbbbbbbbbbbb <Enter>
(gauss) $x' <Enter>
AAAAAAA BBBB BBB
(gauss)

```

Notice that the double quotation mark (') is different from two times the single one (') but they look similar in the file and are at the origin of many mistakes, especially when you import programs.

Remark that a statement is not always executable (example: the declaration of a matrix).

A **program** is a structured set of statements that are run together. In the example below, the programs will be in the *Edit mode*. Consequently I do not put the <Enter>

key at the end of each line nor the GAUSS prompt (**gauss**) at the beginning anymore. For example

```
x=10;  
y=x*3;  
y;
```

and Run it, we obtain in the result in the GAUSS *Command window*

```
(gauss)run c:\gauss\myprog.pgm  
30.000000  
(gauss)
```

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).

GAUSS allows you to read or 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.

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 semicolon. 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
```

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

```
nam_1
_name1
n_a_m_e
```

Remark that GAUSS does not distinguish between uppercase and lowercase, except inside double quotes "...". However, one cannot begin a variable with a number.

```
(gauss) 3y=10 <Enter>
3y=10
^
(0) : error G0063 : Operator missing
1 error(s)
(gauss)
```

The error message informs the user of what is considered an error and where it happens (^). In the Edit mode, error messages often inform about the line that causes the problem. This is very helpful but sometimes misleading. For instance, if you forget a semicolon at the end of a program, GAUSS could report that the problem occurred to a command used several lines after the omission of the semicolon.

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 2 variables for which you have 3 observations. The easiest way to proceed is to type

```
(gauss) Edit mydata.dat <Enter>
```

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

```
1 1
52 54
44 55
```

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.dat`. Naturally such an ASCII file can also be created using an external editor or as the output of various software packages (Excel, RATS,).

The question is now how we can read this small data set so that we will be able to use it in GAUSS. Remember that GAUSS works 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[3,2] = mydata.dat; /*loads the data mydata into the matrix x */
x;
```

which produces the result

```
1.0000000 1.0000000
52.000000 54.000000
44.000000 55.000000
```

while without specifying the rows and the columns of the matrix `x`

```
load f[] = mydata.dat;
f;
```

would give

```
1.0000000
1.0000000
52.000000
54.000000
44.000000
55.000000
```

but if we add the reshape command `f2 = reshape(f,3,2)` we obtain the same result.

`Load` accepts path names if your file `mydata.dat` 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 (`*.dat`, `*.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`) and rename it if you wish.**³ You can use the function `rows(x)` to find out if all the data have been loaded and the function `reshape(x,n,k)` to reshape the $nk \times 1$ vector `x` into the matrix you want to work with.

5 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 that 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:

The command `LET` thus creates matrices.

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

When brace brackets '`{}`' are used, inserting commas in the list of constants instructs GAUSS to form a matrix, breaking the rows at the commas. If brace brackets 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

³In the recent versions of GAUSS, it is now possible to load EXCEL files of the `.xls` type directly. Indeed in order to read these files there exists an `Import` command that I don't see it here.

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. The confusion between brackets '()' , brace brackets '{}' and square brackets '[]' is quite often at the origin of silly mistakes using GAUSS (at least it's my case).

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

```
x = {1 2 3, 4 5 6};
Let x[2,3] = 1 2 3 4 5 6; /* I personally prefer this one but I don't know
why! */
Let x = { 1 2 3, 4 5 6 };
```

and create a 2×3 matrix **x**

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

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];
```

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

Table 1: Examples of LET

<code>let x = 1 2 3 4 5 6;</code>	Shape of x
<code>let x = 1,2,3,4,5,6;</code>	Column vector 6x1
<code>(let) x = {1 2 3 4 5 6};</code>	Column vector 6x1
<code>(let) x = {1 2,3 4,5 6};</code>	Row vector 1x6
<code>let x[3,2] = 1 2 3 4 5 6;</code>	Matrix 3x2
<code>let x[3,2] = 1, 2, 3, 4, 5, 6;</code>	Matrix 3x2
<code>let x[3, 2] = 5;</code>	Matrix 3x2

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 point or the dot `'.'` indicates "all columns". The resulting `x1` is a 1×3 vector

$$x1 = (\ 1 \ 2 \ 3 \)$$

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 manuals, the full list in the Appendix 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
SHOWS(x)	returns both number of columns and 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
SORTC(x,c)	sort the c^{th} column of 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.
- REV(x) reverses the order of rows of a matrix.
- VEC(x) stacks columns of a matrix to form a single column.
-
- 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.

Also note the following important special matrices where most commands are self explaining:

<code>I=EYE(2)</code>	an identity matrix of dimension 2
<code>E=ONES(10,1)</code>	a 10 by 1 matrix of ones
<code>J=ZEROS(15,2)</code>	a 10 by 2 matrix of zeros
<code>T=SEQA(1,1,100)</code>	a 100×1 vector with a sequence starting at 1 with increment of 1
<code>Z=RNDN(10,2)</code>	a 10×2 matrix of standardized 0 mean normal random numbers
<code>Z=RNDU(10,2)</code>	a 10×2 matrix of uniform numbers between 0 and 1

5.1 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	+	<code>y = x + z;</code>
Substraction	−	<code>y = x − z;</code>
Matrix Multiplication	*	<code>y = x * z ;</code>
Division or linear equation solution	/	<code>z = b/A;</code> where A and b are scalars
Element by element multiplication	.*	<code>y = x .* z;</code>
Element by element exponentiation	.^	<code>y = x .^z;</code>
Kronecker Product	.*.	<code>y = x .* . z;</code>
Horizontal direct product	*~	<code>y = x *~y;</code>
Transpose operator	'	<code>y = x';</code>
Vertical concatenation		<code>y = x z;</code>
Horizontal concatenation	~	<code>y = x ~y;</code>

Gauss obviously includes usual scientific functions such as

Abs(x)	returns absolute value of x
Cos(x)	computes cosine
Sin(x)	computes sine
Exp(x)	computes the exponential function of x
Ln(x)	computes the natural log of each element of x
Log(x)	computes the $\log_{10}$ of each elements
Pi	returns π
Sqrt(x)	computes the square root of each elements

Other operators/commands are given and detailed in the *Command Reference* manual or the Appendix. 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** if 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)**, the rank using **RANK(x)**, the Cholesky decomposition by **Chol(x)** while **EIGV** computes the eigenvalues and the eigenvectors of a general matrix. Here is a small program which computes these elements. I do it in Command mode in order to stress the different steps:

```
(gauss) h=eye(2); /* create a 2 dimensional identity matrix */ <Enter>
(gauss) h[2,2]=0; /* put a 0 on the right lower corner to */ <Enter>
(gauss) h <Enter>
1.0000000 0.0000000
0.0000000 0.0000000
(gauss) d=det(h); r=rank(h); d~r; <Enter>
0.00000000 1.0000000
(gauss) {l,v}=eigv(h); <Enter>
(gauss) l /* print eigenvalues */ ; <Enter>
0.00000000
1.0000000
(gauss) v /* print eigenvectors */ ; <Enter>
0.00000000 1.0000000
1.0000000 0.0000000
(gauss)
```

5.2 Comparing matrices, vectors, scalars and strings

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 $z = x \text{ lt } y$;
- Not equal: $z = x \neq y$ or $z = x \text{ ne } y$;
- Greater than: $z = x > y$; $z = x \text{ gt } y$;
- Greater than or equal to: $z = x \geq y$; or $z = x \text{ ge } y$;
- Equal to: $z = x == y$; or $z = x \text{ eq } y$;
- Less than or equal to: $z = x \leq y$; or $z = x \text{ le } y$;

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`.

Most of the logical operators apply to numbers as well as characters and strings. In this latter case you must use the $\$$ -sign in front of the operator. Compare $\$+$ and $\$\sim$ in the following example for concatenating strings.

```
(gauss) y = ''try''$+'me''; y; <Enter>
tryme
(gauss) y = ''try''$\sim''me''; y; <Enter>
      try      me
```

In logical operators we will use $x \$== y$ for comparing characters and strings (see Section 5.3).

5.3 Conditional branching

The syntax of the full IF statement is:

```
if condition1;
  dothis1;
elseif condition2;
  dothis2;
elseif condition3;
  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

In the following example I simulate 50 observations from a $N(0, 1)$ distribution and I calculate the empirical mean, the theoretical one being 0. Then the computer returns a sentence depending on the sign of that estimate.

```
new; /* erases everything in memory */
cls; /* clears the screen */
```

⁴Unconditional 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.

```

r=rndn(50,1); /* generates the numbers */
m=meanc(r); /* takes the mean */
if r < 0;
    print '' The empirical mean is negative'';
else;
    print '' The empirical mean is positive'';
endif;

```

A second example illustrate the use of branching with logical operators for strings.

```

new;
cls;
a=''moitavoiture'';
b=''moitavoitures'';
if a $== b;
    print ''OK boy, no mistakes'' ;
else;
    print '' there is a spelling mistake'';
endif;

```

5.4 Dynamic modeling and creation of lags

Quite often one needs to create lags or first differences of time series and panel data. In these cases

```
y1=lag1(y)
```

takes the first lag of y . For higher lag length we need to use

$w=\text{lagn}(y,p)$ where p stands for the lag we take. For instance, we would use

```
y4=lagn(y,4)
```

to take the fourth lag of y and we call that variable, say, $y4$. First differences are obtained by the statement

```
Dy=y-lagn(y,1);
```

Notice that it is necessary to take the submatrix from observation 2 to the last observation to avoid the first missing value. The instruction `Recserar` computes autoregressive recursive series and is helpful when simulating variables.

6 Examples: Ordinary least squares estimator (OLS)

At this point, it is useful to take a look at some practical examples. We will consider the OLS estimator because it's one of the easiest illustration. Notice that OLS functions already exist in GAUSS (see section on modules).

6.1 Linear regression: Notations

We observe one dependent variable y and k independent variables (including the intercept) X for T periods, $t = 1 \dots T$. 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:

$$y = X\beta + \epsilon \quad (1)$$

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

$$\hat{\beta} = (X'X)^{-1}X'y \quad (2)$$

where $\hat{\beta}$ is a vector of k estimated coefficients obtained thanks to the OLS estimator. The residuals are:

$$e = y - \hat{y} \quad (3)$$

The unbiased estimator of the variance σ^2 is:

$$s^2 = \frac{e'e}{T - k} \quad (4)$$

The variance-covariance matrix of the estimated coefficients is:

$$V(\hat{\beta}) = s^2(X'X)^{-1} \quad (5)$$

Student's t tests are performed using:

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

where SE_i is the standard error of coefficient i . We compute the two sided p - values using the complement of the cumulative density function of the Student's t distribution `CDFTC` although the standard normal approximation with `CDFNC` could be considered when

$T > 30$. One sees that there is no need to encode the critical values from the tables. The R^2 is given by

$$R^2 = 1 - \frac{SSE}{SST} \quad (7)$$

where SSE and SST are respectively the sum of squared residuals and the total sum of squares.

6.1.1 OLS Program

We regress two variables with forty observations I choose at random as follows:

$$\begin{aligned} x_t &\sim U(0, 1) \\ \varepsilon_t &\sim N(0, 1) \\ y_t &= 0.5 + 0.8x_t + \varepsilon_t, \quad t = 1 \dots 40 \end{aligned}$$

Figure 3 reports a small program that generates these series and computes the different OLS statistics. Figure 4 reports the results. It's not possible to print directly a matrix that mixes characters and numbers. For my program that implies I cannot create a matrix which concatenates horizontally the label of the variables and the numbers we have obtained (coefficients, standard errors...). Different means exist to circumvent the problem. The **FTOCV** instruction I have used is a **GAUSS** command which converts a matrix containing floating point numbers into a matrix containing the text representation of each number. In a few words, **FTOCV** translates numbers into text and the outcome can be printed thanks to **\$**. Other commands are available to improve the output. In practice you do not always need to have a neat output. In this case the following statement does the job and gives you the most important information but without the name of the column in front of the estimates you see on my OLS output.

```
PRINT b~sd~t~p_val;
```

Instead of translating numbers in to strings using **FTOCV**, practitioners often use the **PRINTFM** and **PRINTFMT** commands. Their role is to print matrices which mix both characters and numeric values. The difference between these two is that the former allows to add precise format of the different columns. But for the sake of simplicity, let me introduce the latter command. The few lines create the desired output.

```
lab = ''constant''|''b1''; /* create a vector column with labels*/
```

```

GAUSS-Edit
File Edit Search Font Options Window Help
Debug Run > c:\gauss\try.pgm
Save Edit > c:\gauss\try.pgm

NEW; CLS;
/******
/* Generate the data : we fix a seed to recover the same generating process*/
/******
Seed1=12567; seed2= 4555;
eps= rndns(40,1,seed1); x1=rndus(40,1,seed2); y=0.5 + 0.8*x1 +eps;

/******
/* Estimation */
/******
x=ones(rows(x1),1)~x1; 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;
r2= 1 - (sumc[e^2] / sumc[ (y-meanc[y])^2]);
p_val = 2*cdftc(abs(t),rows[x]);          /* two sided p value */

/* In this example, one transforms matrices into characters for the output */

VARname = ["X" $+FTOCV[Seqa[0,1,2],1,0]] $+ " ";
b = FTOCV[b,1,3]; sd = FTOCV[sd,1,3]; t=FTOCV[t,1,3];p_val = FTocv[p_val,1,3];

print "Var      "$~"Est. coeff.  "$~"standard deviation  "$~"t-stats  "$~"p value";
print "-----"-----"-----"-----"-----";
print $varname$+ "      "$+b$+ "      "$+sd$+ "      "$+t$+ "      "$+p_val ;
print "-----"-----";
print "R-squared " r2; print "Number of observations " rows[y];

```

Editing c:\gauss\try.pgm L:6 C:35

Figure 3: OLS program

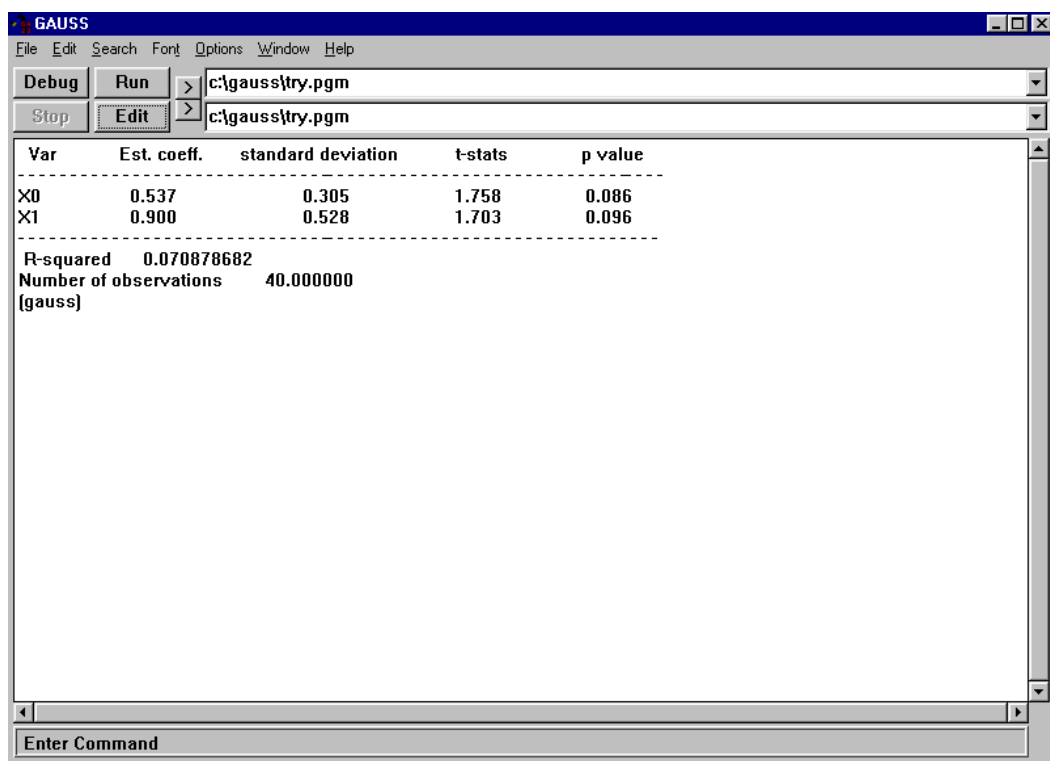


Figure 4: Results

```
var = lab~b~sd~t~p_val ; /* concatenate the name and the variables*/
```

Notice that `var` cannot be printed using either `Print var` or `Print $var`, but the command beneath can.

```
let m[1,5] = 0 1 1 1 1 ; /* define the type of variables */  
                                /* 0 for character and 1 for numeric*/  
y=printfmt(var,m);           /* print the mixed matrix */
```

7 Some tricks when writing out programs: personal point of view

This section gives some hints when writing GAUSS programs. One of the most important and powerful tool consists in writing procedures. That approach will be developed in the next section.

7.1 $\hat{O}$ Memory

Make sure to put a 'New' command at the beginning of a program in order to erase the value of the variables that are stored in GAUSS during the session. For instance you could have the feeling that nothing has changed after running a program although you have modified several lines. That sometimes can be explained by a mistake you have made in your new program. GAUSS doesn't carry out the new lines you have modified and reports the value of the variables of a former program it has kept in memory.

The `CLS` command clears the screen.

If you run into the error message "Unsufficient workplace memory" you can increase the maximum workspace in the `GAUSS.CFG` file. Note that increasing this number decreases the available memory for other programs running under Windows. It may also increase the amount of time it takes to start up Gauss. Hence, do not increase the workspace if it is not necessary.

7.2 Annotation and comments

Don't do like I do that is to say, comment lines of statements and explain the aim of the program at the beginning. It's very helpful when you come back to run or modify an "old program" (it's amazing to see how it's easy to forget the details of a program we spent weeks to write out) or when you give your routines to someone else.

7.3 Format

Use **FORMAT** command to set the output format for matrices, string arrays, and strings printed. Compare for instance the **h** two dimensional matrix given in Section 5.1 with

```
(gauss) Format /rd 5,0; <Enter>
(gauss) h; <Enter>
  1 0
  0 0
(gauss)
```

This is just an example with one set of specific options. See manuals or on-line help for additional for many others.

7.4 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 such as in

```
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**;

7.5 The Con Stuff

It is sometimes useful to use the **CON** command whose purpose is to request input from the keyboard (console) and returns it in a matrix. Imagine for instance that you want to have the OLS estimate given above for different values of **T**, the number of observations (see my program **try.pgm**). The first lines may be replaced by

```
CLS;NEW;
```

```

print '' How many observations do you want for the OLS simulation'';;
n = con(1,1);
Seed1=12567; seed2= 4555;
eps= rndns(n,1,seed1); x1=rndus(n,1,seed2); y=0.5 + 0.8*x1 +eps;

```

Once you run in, the question after the `Print` will appear on the screen and you can type the number you wish after the (?). The double semicolon allows to continue to print on the same line.

8 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 it 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 or function, and any user defined function or procedure can be used *within* a procedure. A procedure looks like

```

proc (number of returns) = NAME(arg list);
  local local variables;

  /* Proc definition goes here. Other
  ** procs, functions, globals, etc.
  */ may be called and referenced
  retp(return args);

```

`endp;`

We see that 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.

We know illustrate the building and the use of procedures with a series of small examples.

8.1 Example 1

Let us first develop a procedure that estimates a linear regression model and returns the OLS estimates, their standard errors and t-values (see the formulae given above). The first lines show how to draw at random the underlying process and how to execute the procedure which is defined in the last 10 lines.

```
NEW;CLS;
Seed1=12567; seed2= 4555;
eps= rndns(n,1,seed1); x1=rndus(n,1,seed2); y=0.5 + 0.8*x1 +eps;
x=ones(rows(y),1)~x1;
{b,sd,t}=regress(x,y);
print ''Estimated coefficients '' b;
print ''Standard errors '' sd;
print ''t-stats '' t;

Proc (3)=regress(x,y); /* here starts the procedure */
  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;
```

Comments:

- 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 **regress** procedure for the arguments that are passed to the procedure when **regress** is called.
- We then define the local variables.
- **rows**, **cols**, **sqrt** and **diag** are global variables known by GAUSS.

- 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 $\hat{\beta}$ (**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.
- The arguments of procedures are in brace brackets before the name such as **{b,sd,t}=regress(x,y**

8.2 Example 2

The following small procedure gives the size, namely the number of columns and rows, of a matrix.

```
new;cls;
proc(2) = dim(x);
  local r,c;
  r = rows(x);
  c = cols(x);
  retp(r,c);
endp;
```

which can be used with any matrices, for instance

```
x= rndn(50,2);
{d1,d2} = dim(x);
print ''This matrix has '' d1 '' rows and '' d2 '' columns '';
```

8.3 Example 3

The following small procedure compare the mean and the median of a matrix in order to know whether it is symmetric or not.

```
new;cls;
proc(2) = stat(x);
  local me, med;
```

```

me = meanc(x);
med = median(x);
retp(me,med);
endp;

```

That can be run for example on

```

x= rndn(25,1);
{d1,d2} = stat(x);
print ''The mean of x is '' d1 '' and the median is '' d2 ;

```

8.4 The Functions using FN

When a procedure consists of one line and returns a single argument one can use a function such as

```
FN fn_name(args) = code_for_function;
```

For instance, let's imagine we want to evaluate the value of the function $y = 14x^3 + 2x^2 + x + 17$ for different values of x and that operation is likely to be repeated several times in your program. Then it's easy to create a function, let's call it `trois` such that:

```
(gauss)FN trois(x) = 14*x^3 + 2*x^2 + x + 14; <Enter>
```

which can be evaluated for $x = 0.698$ to give

```

(gauss)y = trois(0.698); <Enter>
(gauss)y <Enter>
20.433365
(gauss)

```

The good thing is that you can also call this function for a vector of the argument, x in this case. Indeed, instead of getting one point of $y = f(x)$ let us consider the case where we want to have different values of that function, maybe to draw a picture. Then we simply can use something like

```

(gauss)FN trois(x) = 14*x^3 + 2*x^2 + x + 14; <Enter>
(gauss)x=seqa(-2,0.1,41); /* x= -2,-1.9.....2 */ <Enter>
(gauss)y = trois(x); <Enter>

```

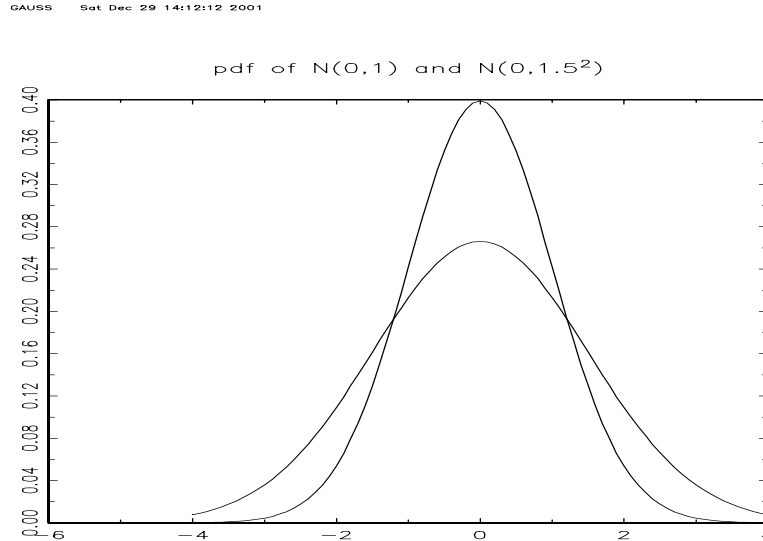


Figure 5: pdf's of Normal distributions

The function can be plotted.

Notice there already exist such functions in GAUSS. For example

```
y=pdfn(x)
```

reports the theoretical probability density function (*pdf*) of the standard normal distribution (*i.e.* $N(0,1)$), namely the value of **y** for different values of **x** using

$$y = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

Let us consider now the more general $N(\mu, \sigma^2)$ *pdf*. We can create a new function

```
(gauss) FN mynorm(x,m,s) = 1/(sqrt(2*pi)*s)*exp(-0.5*((x-m)/s)^2);
```

The following figure compares the `y1=pdfn(x)` command and the one called with

```
(gauss) y2 = mynorm(x,0,1.5);
```

namely a centered normal distribution but with a standard deviation of 1.5 instead of 1. In both cases, the x-axis is generated by `x=seqa(-4,0.1,81)`, i.e. a sequence of numbers from -4 to 4 by increments of 0.1. We'll see in next section how to get these graphs.

9 Libraries

There are several GAUSS libraries that solve many standard statistical and econometric problems such as maximum likelihood estimation in a library called **Maxlik** or the library **Arima** for time series modeling. These libraries are actually packages composed by many GAUSS source codes programmed by somebody else. The GAUSS library is thus a *text file*, the extension must be `.lbg`.

To use these libraries, the information about the different files that constitutes the source code must be read by GAUSS. This is done by the command

```
Library libname;
```

that can be inserted in the beginning of your program or at the GAUSS prompt. Notice that the extension `.lbg` is not needed. For instance, the statement

```
Library maxlik;
```

will say to Gauss that you want to use the program codes that are specified in the file `maxlik.lbg` located in the directory `c:\gauss\lib`. The programs themselves are located in the directory `c:\gauss\src`. If you don't have all the manuals for the different libraries (not included in the two GAUSS manuals), you can read the `.lib` file to see what procedures are proposed and to look at the `.src` source code to know how to introduce the parameters.

Although you will not need all the libraries on a day to day basis, one of the most important one is the one that allows to draw graphs. Note that together with the usual GAUSS commands we have seen so far for computing means, sum, doing branching..., the graphic library is the only one available with the core of Gauss. The other libraries often need to be bought separately. You can access to the graph library by the statement

```
Library pcgraph;
```

The next section explains how to draw graphics. We will call other libraries in the section about estimation by maximum likelihood.

10 Graphics in Gauss

One attractive feature of GAUSS is its ability to draw high quality graphs in two or 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. As we have just seen, 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. The two magic lines are then

```
library pgraph;  
GRAPHSET;
```

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'';
```

which 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. There are codes for defining fonts and special characters for titles inside the `''''`. For the graphs I plotted in the previous section I use the command

```
title('pdf of N(0,1) and N(0,1.5[2])')
```

to put the standard deviation at power 2. `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. 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 $(\text{\#row} \times \text{\#col})$ 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: **eps** (postscript), **pic** (Lotus), **hpgl** (plotter) and **pcx** (bitmap). I would favor the **eps** format for importation into Scientific Wordplace.

10.1 Example 1

The first example plots the two simulated series we have already used for OLS estimation.

```
library pgraph;
graphset;
n=100; Seed1=12567; seed2= 4555;
eps= rndns(n,1,seed1); x1=rndus(n,1,seed2); y=0.5 + 0.8*x1 +eps;
t = seqa(1,1,n);
title(''Simulated data set'');
_plctrl= { 0 0} ; /* connect points*/
_pltype = { 6 2}; /* type of curves */
_pbox= 15; /*draw a box around the graph*/
```

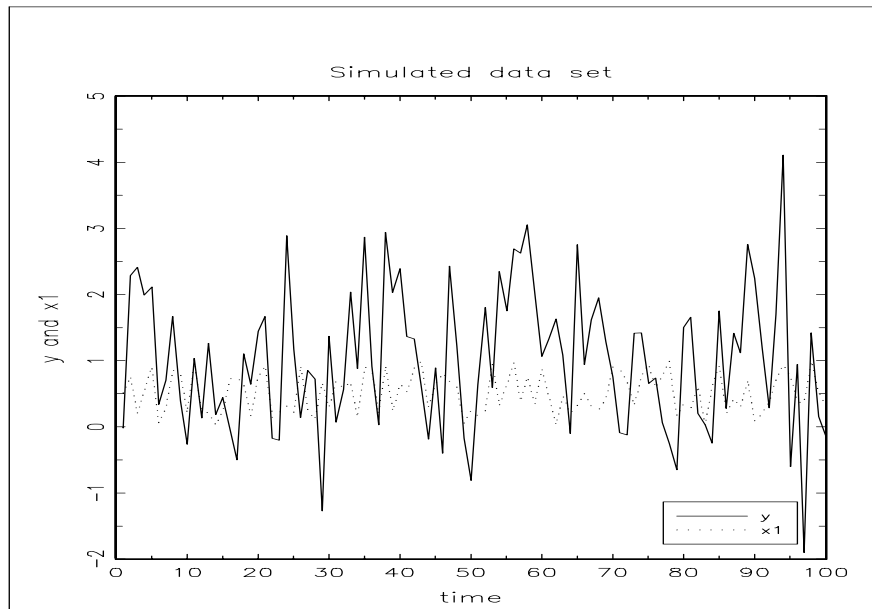


Figure 6: Example 1

```
_pdate = ''; /* to do not print the date */
_plegctl = 1 ; /* create a legend box */
_plegstr = 'y \000x1\000'; /* put names for the legend */
xlabel('time');
ylabel('y and x1');
_protate= 1; /* you probably don't need this */
xy(t,y~x1);
```

Remark: This is from the *.eps file created in my old GAUSS 3.2.13 Dos version.

10.2 Example 2

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 variables and first differences are computed. Finally a page with four graphs is created.

```

library pgraph;
graphset;
n = 400;
seed1=57;e=rndns(n,2,seed1);/*generating 2 pseudo normal random data*/
obser = seqa(1,1,n); /*generate a deterministic series 1,2,...,n */
y1 = CUMSUMC(e[.,1]); /*cumulate the y1 series to get a random walk */
y2 = recserar(e[.,2], e[1,2:2],1) ; /* another way to do it with y2*/
y = y1~y2;
y1 = lagn(y,1); /* generate lagged variables */
yd = y-y1; /* generate first differenced series */
begwind; window(2,2,1);
_protate=1; /* you probably don't need this */
setwind(1); title('Figure 1'); /* 1st figure */
xlabel('Observations'); ylabel('Variables y - level');
xy(obser,y);
setwind(2); title('Figure 2'); /* 2nd figure */
xlabel('Observations'); ylabel('Variable y1 - 1st diff. ');
xy(obser,yd[.,1]);
setwind(3); title('Figure 3'); /* 3rd figure */
xlabel('Observations'); ylabel('Variable y2 - 1st diff. ');
xy(obser,yd[.,2]);
setwind(4); title('Figure 4'); /* 4th figure */
xlabel('Random numbers'); ylabel('Frequencies. ');
{b1,m1,freq1} = HISTP(y[.,1],25);
endwind;

```

11 Loop statements: While, Until and For

Often one needs to do the same operation over and over again. This can easily be done using a loop. The format for the loop statements are

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

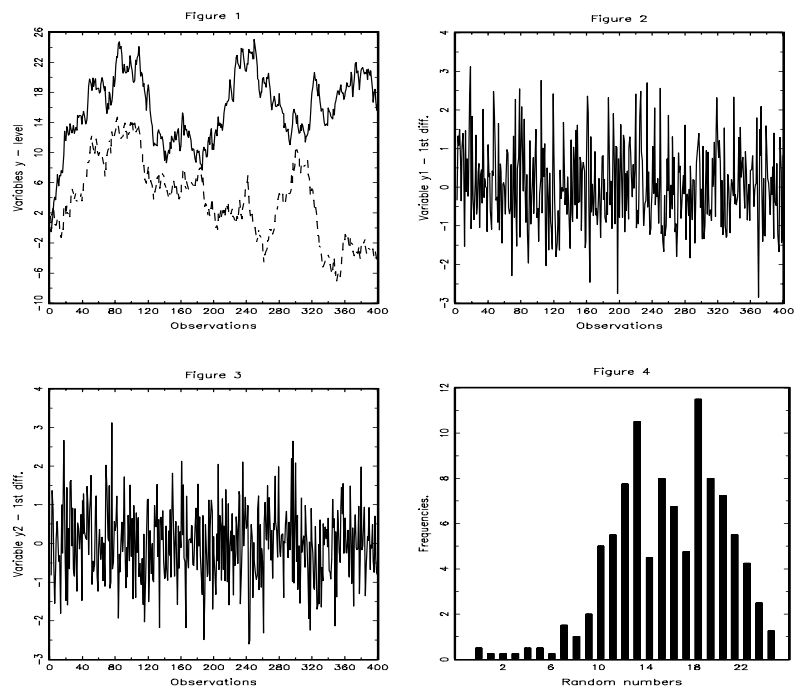


Figure 7: Example 2

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. For instance, here are a small example on the print of an index (useful during simulations). Since version 3.2.35, GAUSS now also offers a **FOR** loop construct. The format is

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

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. **Wait** stops the program within the loop until you press a key. It's convenient to employ **WAIT** when you want to look at intermediary results within a loop in order to be sure that everything is fine and, if not, to check where the mistake comes from.

We illustrate three equivalent ways to obtain the values of **i** within the loops, i.e. **i=1,2,3,...10**.

For	Do Until	Do While
<pre>for i (1, 10, 1); i; endfor;</pre>	<pre>i = 0; do until i >= 10; i=i+1; i; endo;</pre>	<pre>i = 0; do while i <10; i=i+1; i; endo;</pre>

12 Small-Sample Inference: Resampling Methods

This section presents two econometric tools for which the GAUSS programming language is extremely well suited (the best one in my opinion) as it is fast and it heavily makes use of matrix operations and loops. We discuss both Monte Carlo experiments and the bootstrap. They solve different aspects of small sample inference. Monte Carlo simulations are used to analyze the behavior of different estimators or tests statistics. Exemple are: what are the small sample properties of the 2SLS, what happens if we use the LSDV in panel data while the model is a random coefficient one, what is the behavior of unit root tests in the presence of heteroscedasticity, what are the consequences of considering too many instruments in the GMM, does nonlinearity change the distribution of the normality test...The bootstrap is used to get the small sample distribution of the chosen estimator or test statistics when the later are difficult or impossible to obtain: distribution of a ratio, of quantiles or of impulse responses.

12.1 Monte Carlo Simulations

Monte Carlo experiments are powerful techniques very often used in applied econometric analyses to asses the small sample behavior of estimator and test statistics. A Monte Carlo experiment consists in different parts.

- 1) Define the issue to analyze,
- 2) The data generating process (DGP) where your develop your known process,
- 3) Generating numbers at random,
- 4) Estimation and test statistics for one replication of the loop. Use of procedures,
- 5) Storage and computation of summary statistics over the M replications.

To illustrate these different steps, let us consider the relationship between two independent random walks (with drifts). Since Yule (1926) but more recently from Granger and Newbold (1974), Engle and Granger (1987), Phillips (1996) we know that if we regress these variables on each other, there is a tendency to obtain spurious regressions.

This phenomenon is characterized by values of t ratios for which we would reject the null hypothesis of no relationships at any sensible significance levels. Moreover the R-squared are high because of the common stochastic trends.

For the DGP, I generate the following independent bivariate process such as

$$\begin{cases} y_t = 0.3 + \rho_1 y_{t-1} + \varepsilon_{1t} \\ x_t = 0.5 + \rho_2 x_{t-1} + \varepsilon_{2t} \end{cases}, \quad \begin{pmatrix} \varepsilon_{1t} \\ \varepsilon_{2t} \end{pmatrix} \sim NIID \left[\begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right]$$

and I estimate the static relation $y_t = \alpha + \beta x_t + u_t$ using the OLS estimator. We are interested in the estimated coefficient $\hat{\beta}$, in the Student's t associated with that coefficient as well as in the value of the R^2 . I consider successively two stationary processes with $\rho_1 = \rho_2 = 0.25$ and the spurious regression case where we have two random walks with drift, that is to say $\rho_1 = \rho_2 = 1$.

In the following program, results are obtained by considering explicitly the OLS estimator within the loop. Alternatively and more efficiently, the OLS procedure written above can obviously be used. We must keep a trace of the interesting statistics for the computation outside the loop. It's why we should stack these variables and initialize their corresponding names. It is also wise to simulate more observations than the ones we need (+ 50 in this example) and drop these first observations for the computation of the statistics. That decreases the impact of the initial conditions.

```
new;
T = 100 + 50; /* 100 observations + 50 presample values */
N= 10000;      /* the number of replications is 10000 */
seed1=124564; /* fix the seed */
rstat=0; bstat=0; size=0; /* initialize statistics */
i=1;          /* I start the loop */
do while i<=n;
e=rndns(t,2,seed1); /*generating 2 pseudo normal random data*/

/* generate 2 stationary autoregressive processes,
**          rho=0.25 with drifts.
/*          Replace 0.25 by 1 for random walks
rho=0.25;
y1= recserar(0.3 + e[.,1],0, rho);
y2= recserar(0.5 + e[.,2],0, rho);
y1= y1[51:t,.]; /*discard the first 50 observations */
```

```

y2= y2[51:t,.];
x = ones(t-50,1)~y2;
xxi=invpd(x'x); b=xxi*(x'y1); e=y1-x*b;
s2=e'*e/(rows(x)-cols(x));
sd=sqrt(diag(s2*xxi));
tstud=b./sd;
t2= tstud[2,1]; /* take the t-stat for the slope coeff */
bols2 = b[2,1]; /* idem */
r2= 1 - sumc(e^2) / sumc((y1-meanc(y1))^2);

size = sumc(abs(t2).> 1.96) + size ;
rstat =rstat|r2;          /* stack r-squared */
bstat = bstat|bols2;     /* stack coefficients */
i=i+1;
endo;

output file = es3.res on; /* or reset;*/
print; format /rz 10, 5;
'' *****
* RESULTS *
*****
'';
'' nb obs = '' rows(y1) ;
'' nb replic = '' n ;
'' Size = '' (size*100/n) ;
'' R-squared = '' meanc(rstat[2:n+1,.]);
'' Mean coeff = '' meanc(bstat[2:n+1,.]);
''';

```

Figure 8 shows the output for both the stationary and the non-stationary cases. With two independent random walks we would find a significant relationship at a significance level of 5% in 99.72%.

In the previous example I have generated independent processes. For generating a dependent multivariate distribution with a more general variance-covariance matrix, the Cholesky factorization may be needed. For instance a bivariate normal distribution with $\sigma_{12} \neq 0$ is obtained with

```

sigma= (4~1.5)|(1.5~1); /* var-covar matrix */

```

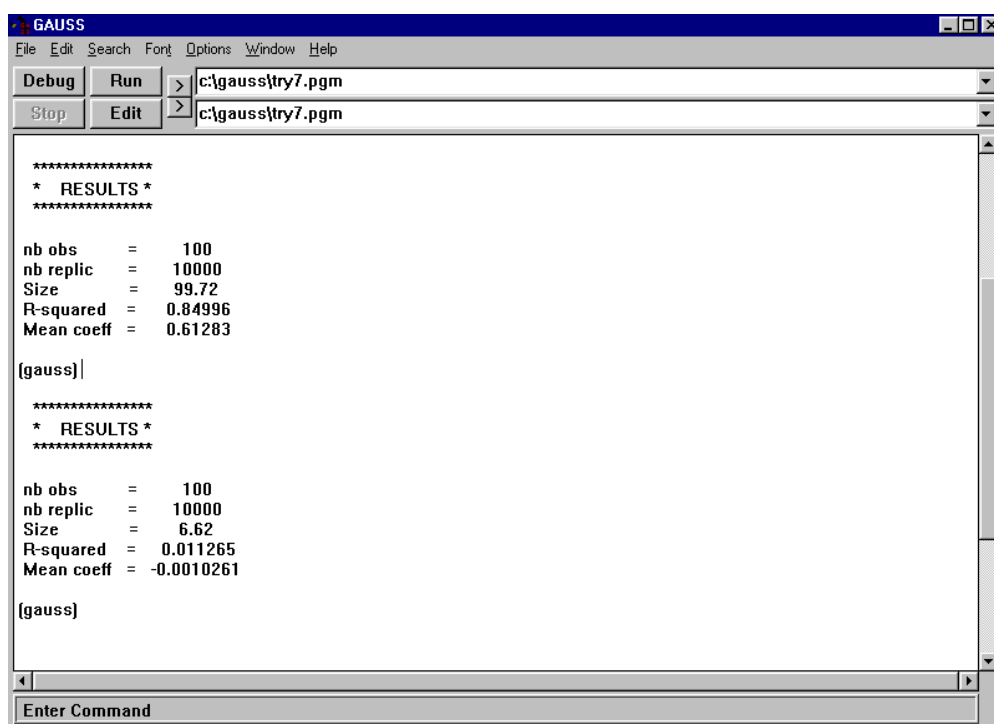


Figure 8: Monte Carlo summary results

```
L = chol(sigma); /* sigma = L'L */
eps = rndns(T+50,2,seed1)*L ;
```

Quite often, the CUMSUMC and RECSERAR commands are not flexible enough for generating dynamic processes and we need to create our own procedures and loops to do it. The next example generates a two dimensional VAR(3), i.e. a bivariate multivariate process such as

$$y_t = \Phi_1 y_{t-1} + \Phi_2 y_{t-2} + \Phi_3 y_{t-3} + \varepsilon_t$$

```
proc simVAR(Z,phi1,phi2,phi3);
  local v, ie;
  v=zeros(t+50,2);
  ie=5;
  do until ie > t+50;
  v[ie,.]=v[ie-1,]*phi1' + v[ie-2,]*phi2' + v[ie-3,]*phi3' + z[ie,.];
  ie = ie+1;
  endo;
  retp(v);
endp;
```

which can be called with the statement

```
y = simvar(eps,phi1,phi2,phi3);
```

for a given bivariate random variable **eps** and three 2×2 coefficient matrices.

Notice that there are basically two types of random numbers generated in GAUSS. These are the standard normal and the uniform [0,1] distributions from which one can create in practice any other distributions (Student's t , Gamma...). As a more simple example, the few following command lines create a general $N(\mu, \sigma^2)$ from the $N(0, 1)$.

```
(gauss)seed1 =5864; <Enter>
(gauss)z = rndns(100,1,seed1); <Enter>
(gauss)m =25; s=4; /* mean=25, std. dev.=4 */ <Enter>
(gauss)y = z*s2 + m; <Enter>
```

In another example I need to generate 5 random real numbers in order to know what group is going to present his homework next week.

```

(gauss) c=rndu(5,1); <Enter>
(gauss) c' <Enter>
0.29569423 0.35256349 0.82680328 0.027466527 0.86649193
(gauss) c2=c*5 <Enter>
(gauss) c2' <Enter>
1.4784711 1.7628174 4.1340164 0.13733263 4.3324596
(gauss) c3=ceil(c2) <Enter>
(gauss) c3' <Enter>
2.0000000 2.0000000 5.0000000 1.0000000 5.0000000
(gauss)

```

The second group will present next week and the week after (not to be announced in advance), then group 5 will present...

12.2 Pseudo-random numbers and the use of the seed

Actually, only the uniform distribution is necessary to compute all other distributions because standard normal random numbers can be obtained by inverting the normal cumulative distribution function which is always between $[0,1]$, i.e.

$$\begin{aligned}
 u &\sim U(0,1) \\
 x &= F^{-1}(u) \sim N(0,1)
 \end{aligned}$$

where $F^{-1}()$ is the inverse of the cumulative distribution function of the random variable of which we want to make random draws. The GAUSS command to compute $F^{-1}()$ the inverse of the cumulative distribution function (*cdf*) of the normal distribution is

```
x = cdfni(p);
```

The following small program does this operation and Figure 9 plots the outcome.

```

library pgraph;
graphset;
u1 =rndus(250,1); u1 = sortc(u1,1);
x1 =cdfni(u1);
Title('Generating N(0,1) random numbers from a U(0,1)');
ylabel('u ~U(0,1)');

```

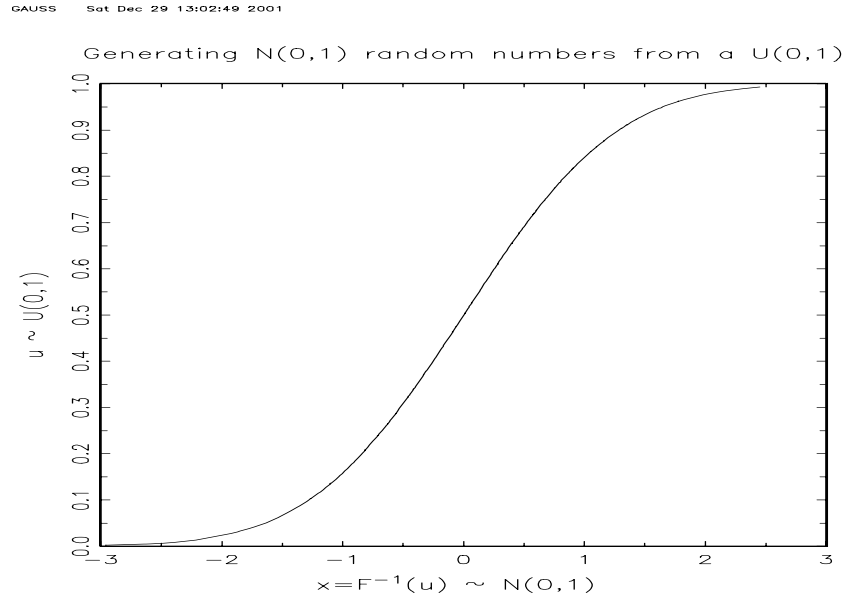


Figure 9: The Normal distribution from the Uniform

```
xlabel('x=F[-1](u) ~N(0,1) ');
_protate=1;
xy(x1,u1);
```

But hopefully the main *pdf* and *cdf* are already implemented in Gauss.

Still remains unsolved the issue about where the $U(0,1)$ comes from. Most software packages use a pseudo-random generator formula which is actually nothing else than a non-linear deterministic function which mimics the randomness of a $U(0,1)$ (See Chapter 5 in Greene). But these formulas need a initial value (the seed) to start generating these numbers. If not stated, the initial value is taken from the clock of the computer. But in simulation experiments we also need to control the sequence of random series which are generated. In my case where I have analyzed the spurious regression issue, I needed to separate the effect of having random walk series to the effect of pure randomness due to differences in the simulated distributions for the error terms. This is the role of the seed command to fix the initial value and then to be able to replicate the sequence of generated series from a simulation to another. That was not important for the previous

example but in practice, empirical sizes from 5% to 7% can be attributed to different sequences and shouldn't be interpreted. Remark that fixing the seed doesn't imply that in a loop, all the numbers drawn at random will be the same (you must check that). But only that if one takes 1000 replications, one will get 1000 different sequences of say 250 observations and that sequence will be reproduced if one opens the computer two days later with the same seed.

12.3 Bootstrap

The bootstrap is a method for estimating the distribution of an estimator or test statistic by resampling ones data. It amounts to treating the data as if they were the population for the purpose of evaluating the distribution of interest. Under mild regularity conditions, the bootstrap yields an approximation to the distribution of an estimator or test statistic that is at least as accurate as the approximation obtained from first-order asymptotic theory. Thus, the bootstrap provides a way to substitute computation for mathematical analysis if calculating the asymptotic distribution of an estimator or statistic is difficult. An example is the computation of the distribution of the long-run elasticity $\theta = (\beta_1 + \beta_2)/\alpha_1$ in the following ADL(1,1) model

$$y_t = \alpha_1 y_{t-1} + \beta_1 x_t + \beta_2 x_{t-1} + \varepsilon_t$$

We give now a simple example of the bootstrapping of the mean and the standard distribution of the mean of a series.

```
/* Purpose:  construct a bootstrap estimate of the sampling distribution
of a sample mean */
new;cls;
n=100;          /* sample size */
b=10000;        /* Set number of re-samples */
et = hsec;
xbar_vec = zeros(b,1);
x=rndn(n,1)*4 + 3;          /* our vector of observation*/
xbar = meanc(x);             /* mean of x */
sdt_x = stdev(x);           /* standard deviation of x */
sdt_xbar = sdt_x/sqrt(n); /* standard deviation of mean of x */
i=1;
do while i<=b;
```

```

index=ceil(rndu(N,1)*N) ; /* Set up re-sampled index */
x_resam =x[index,.]; /* Set up vector of x */
xbar_b = meanc(x_resam); /* Calculating the mean of re-sample */
xbar_vec[i,1] = xbar_b; /* Saving x-bar* in vector */
i=i+1;
endo;
boot_xbar=meanc(xbar_vec); /* Average of the x-bar*'s, the bootstrap estimate*/
sd_xboot=stdc(xbar_vec); /* Standard deviation of the x-bar*'s */
et = (hsec - et)/100;

''Sample size = '' n;
''Number of bootstrap re-samples (b) = '' b;
print;
''Full sample mean = '' xbar;
''Full sample sd(mean of x) = '' sdt_xbar;
print;
''Bootstrap Mean of x = '' boot_xbar;
''Bootstrap Sd of x = '' sd_xboot;
print;

```

13 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).

13.1 Maximum likelihood

Maximum likelihood techniques are quite popular in econometrics because the underlying intuition is simple. It involves choosing values for the parameters that maximize the chance (or the likelihood) of the data occurring. Let us start along Hull (2000, p.374) with a very simple example where ten stocks are sampled at random on a certain day and it is found that the price of one of them declined on that day and the prices of the other nine either remained the same or increased. In this case the best estimate of the

proportion of all stocks with price decline is naturally 10%. To see how the maximum likelihood method works, we must write down the joint probability density function (the other name of the likelihood function actually) that one particular stock declines and the other nine not, that is $L = p(1 - p)^9$ and take the minimum of this function, that is to say to find the value of p whose gradient is equal to zero, i.e. to compute $\frac{dL}{dp} = 0$. Because it's easier to differentiate a linear function, we can reach the same maximum for the log of the likelihood function

$$l \equiv \ln(L) = \ln p + 9 \ln(1 - p). \quad (8)$$

We directly obtain $\frac{dl}{dp} = \frac{1}{p} + \frac{-9p}{(1-p)} = 0$, that is $p = 0.1$. There obviously exist a tractable and exact solution for (8). In other cases numerical methods and algorithms are needed. Let me first introduce a grid search procedure where we feed Equation (8) with different values of p and retain the one which maximize the function. For example,

```
new;cls;
let y[10,1] = 0 1 1 1 1 1 1 1 1 1; /* the data */
i = sumc(y); /* number of ones */
j = rows(y) - i; /* number of zeros */
x = seqa(0.05,0.05,10); /* 10 values of p between 0.5 and 5 */
fn lp(p) =j*ln(p) + i*ln(1-p); /* define the function */
k= lp(x); /* use the function */
x~k; /* print the results */
```

equals

```
0.050000000 -3.4573719
0.100000000 -3.2508297
0.150000000 -3.3597904
0.200000000 -3.6177299
0.250000000 -3.9754330
0.300000000 -4.4140473
0.350000000 -4.9268684
0.400000000 -5.5137213
0.450000000 -6.1790407
0.500000000 -6.9314718
```

(gauss)

where we see that the function is maximized for $p=0.1$. In practical situations a additional search in the neighborhood of 0.1 is needed to have a fine tuning estimation. A more efficient way to proceed is by using the command `sqpsolve()` for minimizing non-linear equations. I have generated 1000 observation between -0.7 and 3, then I take the `ceil` operator that round up toward positive infinity, namely `ceil(-0.3)=0` and `ceil(0.3)=1` for example.

```
new;cls;
y = ceil(rndu(1000,1) -0.3);
i = sumc(y);
j = rows(y) - i;
fn lp(p) = -1*(j*ln(p) + i*ln(1-p)); /* I take (-) because */
/* sqpsolve minimize a function */

stval = 0.5;
{x,f,g,retcode} =sqpsolve(&lp,stval);
```

gives

Value of objective function 619.898448

Parameters Estimates

P01 0.3110

Number of iterations 4

Minutes to convergence 0.00100

(gauss) i

689.00000

(gauss) j

311.00000

(gauss)

We give now the basis of the ML estimators for general cases and we use the algorithms developed for these purposes which are much more efficient than the ones just sketched.

13.1.1 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:

$$\max_{\theta} L(\theta, y) \implies \hat{\theta} \quad (9)$$

- Based on $\hat{\theta}$, the maximized likelihood is $L(\hat{\theta}, y)$.
- 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, $\hat{\theta}$ is such that $s(\hat{\theta}, y) = 0$, which allows the computation of $\hat{\theta}$.
- 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: $IA(\theta) = \lim_{T \rightarrow \infty} I(\theta) / T$

13.1.2 Two important properties of the maximum likelihood estimator

- The ML estimator is asymptotically unbiased and fully efficient.
-

$$T^{1/2}(\hat{\theta} - \theta_0) \rightarrow N_k(0, IA^{-1}(\theta_0)) \quad (10)$$

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

$$\widehat{\theta} - \theta_0 \rightarrow N_k(0, (-H(\widehat{\theta}, y))^{-1}) \quad (11)$$

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

13.1.3 Numerical procedures

- Iterative numerical procedure for θ : $\theta_1, \theta_2, \dots, \theta_N$ 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.

13.1.4 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. Since a fast analytical solution to this

problem exists i.e., $b = \text{inv}(x'x) * x'y$ the only reason to use the numerical solution is for pedagogy. Note that sigma-hat is not the same with OLS and ML. OLS programs usually use the unbiased estimator which differs by $T/(T - k)$ from the ML estimator. The intercept and slopes should be identical. 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 */
NEW; CLS;
library maxlik; /* open de Maxlik library */
maxset;
output file=myml.out on;

/* Generate the data :
** we fix a seed to recover the same generating process
*/

print '' How many observations do you want for the OLS simulation'';;
n = con(1,1);
Seed1=12567; seed2= 4555;
eps= rndns(n,1,seed1)*2; x1=rndus(n,1,seed2); x2 = seqa(1,1,n);
y=0.5 + 0.8*x1 + 0.06*x2 +eps;

/* Maximum likelihood estimation */
stval=zeros(cols(x1~x2)+1,1)|4;
/*start coeff. values to zeros and sig to 4 */
proc loglik(theta,data);
  local xb,x,y,llik,s,s2,u;
  y=data[.,1]; /* dependent variable */
  x=ones(rows(data),1)~data[.,2:cols(data)];
  /* explanatory variable + intercept */
  s=theta[rows(theta)]; /* where to take s */
  theta=theta[1:cols(x1~x2)+1] ; /* where to take the coeff */
  xb=x*theta;
  s2=s^2;
  u=y-xb;

```

```

llik=-0.5*(ln(s2)+u.*u/s2);
retp(llik);endp;

''***** Analytical Solution *****'';?; /* OLS */
call ols(''',y,x1~x2); /* use the OLS procedure in Gauss */
''press a key to maximize numerically...'';wait;?;?;
/* Names of parameters
** _max_ParNames - Kx1 character vector, parameter labels.
*/
_max_parnames = ''const'' | ''b1'' | ''b2'' | ''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 optimization
** _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
** satisfied OPTMUM will exit the iterations.
*/
_max_GradTol =1e-5;
{x,f,g,cov,retcode} =maxlik(y~x1~x2,0,&loglik,stval);
call maxprt(x,f,g,cov,retcode); /* control the output of ML*/

```

Figures 10 and 11 show the results of both the OLS GAUSS procedure and the ML estimation.

13.2 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

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

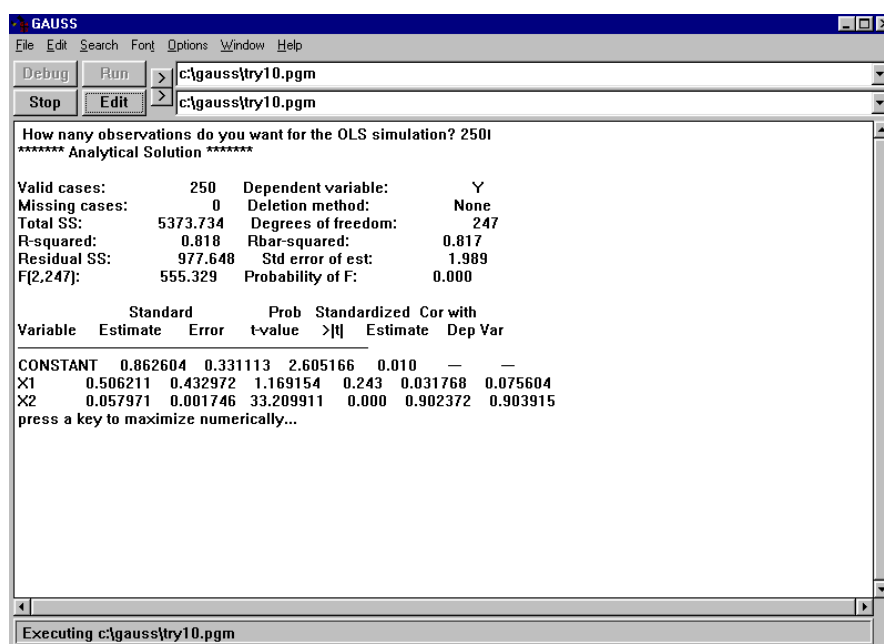


Figure 10: First Part of the Program - OLS Estimates

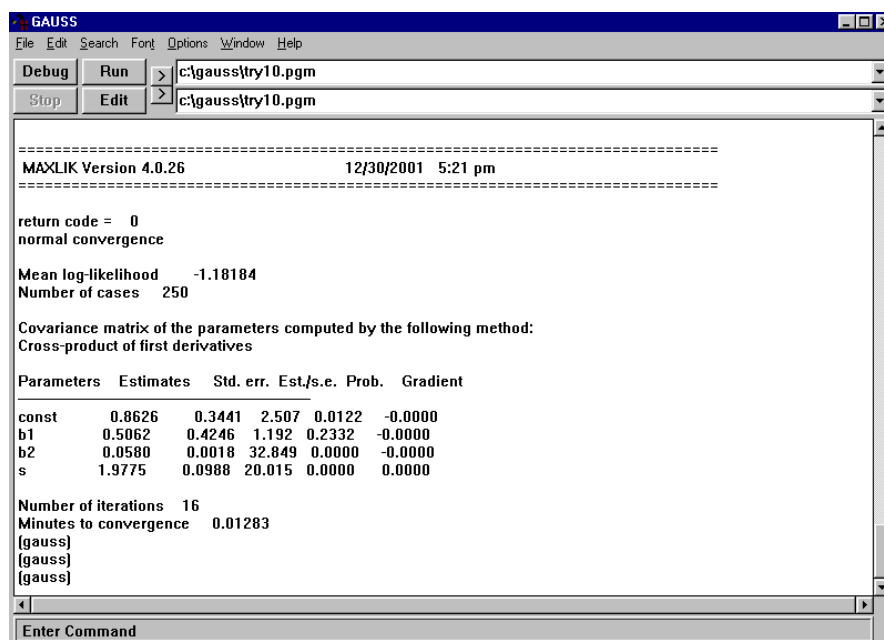


Figure 11: ML Estimation Output

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_t which we have already loaded in a $T \times 1$ vector x_t . 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).